HARDWARE MODELING WITH VHDL SIMULATION

THESIS

Douglas G. Pompilio
Captain, USAF

AFIT/GCS/ENG/88D-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88D-15

HARDWARE MODELING WITH VHDL SIMULATION

THESIS

Douglas G. Pompilio
Captain, USAF

AFIT/GCS/ENG/88D-15

AFIT/GCS/ENG/88D-15

HARDWARE MODELING WITH VHDL SIMULATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

Douglas G. Pompilio, B.S.
Captain, USAF

December 1988

Acknowledgments

This thesis culminates 18 months of study that would not have been possible without the guidance and support received from fellow students and friends. First, I would like to thank Major Joseph DeGroat, my thesis advisor, who's insight into this project helped lead me to the promised land. I appreciate your giving me the freedom to conduct the necessary research, and at the same time, I appreciate the guidance and direction you provided to this sometimes lost thesis student. To my thesis readers, Captain Bruce George and Captain William Bralick, thanks for taking the time to read my drafts and providing comments and suggestions. To my cohorts who live in the VLSI trailer/lab (not an exaggeration for some of these guys), thanks for your ideas, support, and camaraderie.

To Mom and Dad, thanks for teaching me the importance of an education. And lastly (but most importantly), to my family, whose love and support endured 18 months of some very trying times, I want to express my sincere appreciation. Cynthia, Matthew, and Benjamin, I couldn't have done it without you!

Douglas G. Pompilio

Table of Contents

Page

Page

Page

Page

List of Figures

List of Tables

AFIT/GCS/ENG/88D-15

Abstract

This thesis presents a study of the VHSIC Hardware Description Language (VHDL) and its ability to accurately model digital hardware circuits. A brief background of Hardware Description Languages and VHDL is presented followed by a detailed look at VHDL's language features and semantics that support hardware modeling. This information is applied to the design and development of a VHDL simulator that supports a subset of the language. A discussion of the simulator design and implementation issues is presented. theses. ([illegible])

HARDWARE MODELING WITH VHDL SIMULATION

I. Introduction

1.1 Background

The Department of Defense (DoD) formed the Very High Speed Integrated Circuits (VHSIC) program in 1979 to promote the use of high density integrated circuits (ICs) in military electronic systems [Dewe86, Myrv85]. The goal of the VHSIC program is to reduce the design time for developing VHSIC class digital systems. To achieve this goal, a standard means of designing, documenting, and verifying digital systems is needed. These needs motivated the DoD to sponsor the development of the VHSIC Hardware Description Language (VHDL) to become the standard method for describing and documenting electronic and digital hardware devices.

In 1983, the VHSIC program contracted Intermetrics, Inc. to develop VHDL along with an computer aided design (CAD) environment to help design and test VHDL models via software analysis and simulation. The acceptance of the language by the engineering community led to a concerted effort by the government and the Institute of Electrical and Electronics Engineers (IEEE) to further enhance the language and establish it as a standard [Bart88]. The standardization effort began in February 1986 and was supported by experts from engineering, computer, manufacturing, and other technical fields [IEEE88]. This effort culminated in the approval of what is now known as IEEE Standard VHDL 1076-1987.

The development of VHDL not only helps the DoD with its goal of advancing the technology of its military system, but as a standard hardware description language (HDL), VHDL will promote the advancement of technology across a wide spectrum of digital electronic applications. VHDL is touted as being an "extremely important vehicle for the development and insertion of VHSIC technology in the 1990's" [Koda87, Dewe86]. As such, use of VHDL by engineers and circuit designers must begin now to breed familiarity for use as a system specification and design language. Additionally, adding VHDL training to academic curriculums will help make future engineers and circuit designers proficient in VHDL before they enter their profession [DeGr88].

To promote the use of VHDL in an educational environment, research at AFIT was consolidated to develop the AFIT VHDL Environment (AVE) [Cart87]. AVE was developed with the intention of providing all the necessary tools to use VHDL under the UNIX[1] operating system. AVE includes both a VHDL analyzer [Berk88] and simulator along with a graphical interface program [Mate88] (Figure 1-1). The analyzer checks the VHDL source code for correct syntax and converts it into a form that can be efficiently simulated. The simulator allows the designer to test circuit designs by viewing their operation under computer simulation. The graphical interface program lets the user create digital designs graphically, and when fully implemented, will automatically generate VHDL source code from the design layout.

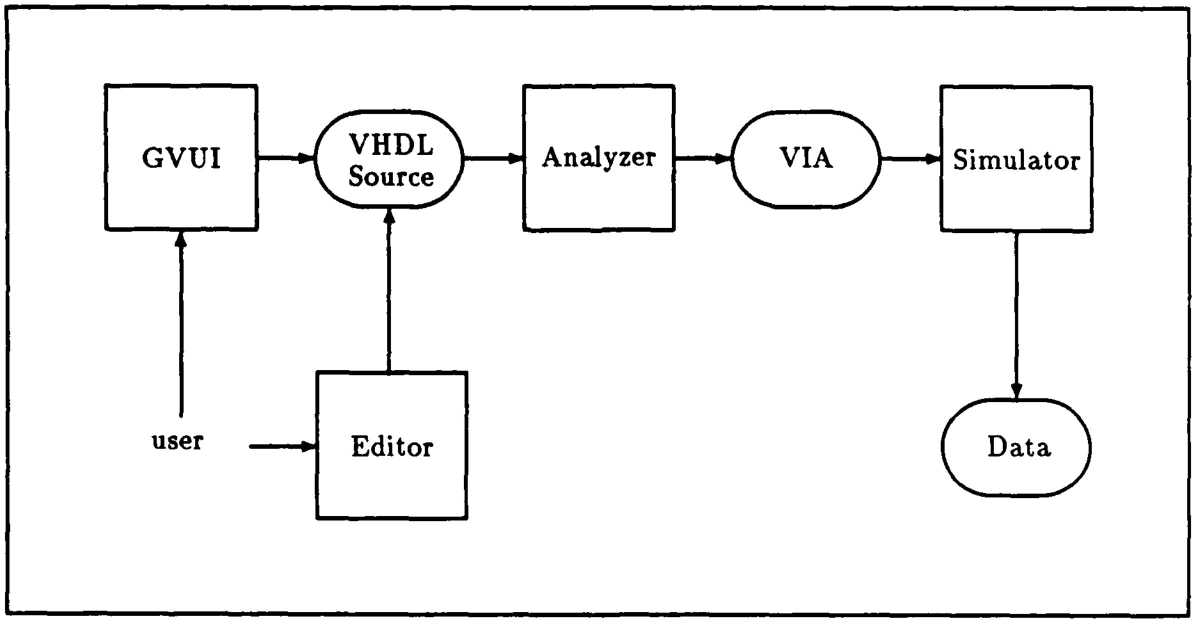

Figure 1-1. AFIT VHDL Environment

[1]UNIX is a trademark of AT&T.

1.2 Problem Statement

The objective of this thesis is to: 1) analyze VHDL in terms of its ability to accurately model digital hardware circuits and 2) design and implement a subset simulator that adheres to the hardware modeling semantics of the language. The simulator implementation will be based upon a prototype VHDL simulator developed by Maj William Lynch [Lync86] and a subset simulator developed by Lt Harvey Kodama [Koda87], both of which worked with VHDL version 7.2. The new simulator will extend the capabilities of the existing simulators and redefine their operation to support VHDL as defined under IEEE Standard 1076-1987. The AVE simulator must be compatible with the other computer aided design tools used in the AFIT VHDL Environment. Lastly, a complete set of documentation must be written to support the simulator's release to universities and other government agencies.

1.3 Scope

This thesis must consider the changes to VHDL 1076 from version 7.2 and analyze the impact of these changes on the AVE simulator. The simulator software development will include updating the current simulator to conform with the new IEEE standard for VHDL, restructuring of the simulator design to improve efficiency, and the addition of critical VHDL features. The final version of the AVE simulator will not support the complete language, but it will provide a full range of functions to support the simulation of realistic designs.

The software will be written using the C programming language and will be implemented on a VAX or an ELXSI computer operating under the UNIX operating system. Although the primary target machine for running the simulator during this thesis will be a VAX or an ELXSI, the software will be developed to allow it to run under any C programming environment.

1.4 Current Knowledge

The AFIT VHDL software simulator has been the topic of two previous theses. In 1986, Maj Lynch began the development of the AVE simulator by generating a preliminary design and a prototype simulator that worked with a subset of VHDL [Lync86]. One of his objectives was to determine the feasibility of simulating Very Large Scale Integrated (VLSI) circuits that were modeled using VHDL. He found the number of individual elements making up a VLSI circuit too great for all components to be efficiently simulated at the same time. He concluded VLSI circuits can be efficiently simulated using an event-driven simulation.

Maj Lynch's success not only proved that simulation of VLSI circuits modeled with VHDL was feasible, but he also paved the way for continued research and development in this area. In his concluding remarks, Maj Lynch stated that the prototype simulator has provided: "1) a proof of design concept for development of a complete VHDL simulator for the UNIX Environment, and 2) an established baseline upon which future research and development efforts can build" [Lync86:94].

In 1987, Lt Kodama extended the work of Maj Lynch and produced a simulator that accommodated the use of a subset of VHDL [Koda87]. Lt Kodama's work was significant in developing a functional AFIT environment that allowed both the analyzer and simulator to work together in providing an integrated tool to analyze and test circuit designs. One of Lt Kodama's objectives was creating a modular design for the simulator that facilitates implementation of the full language. The impact of Lt Kodama's work helps pave the way towards a complete simulator as he successfully produced a design that will readily accommodate future additions or changes to VHDL.

1.5 Approach

The simulator is but one portion of the AFIT VHDL Environment; therefore, the general design philosophy is to implement simulation support for VHDL language features incrementally in coordination with other AVE tool development. Developing the simulator in this fashion allows for comprehensive testing of each feature of the design along with ensuring total compatibility between the simulator and other AVE tools.

Once the software development is complete, comprehensive system level testing will be conducted. At issue are accuracy of the design, proper software implementation, and runtime efficiency. Proper implementation of VHDL modeling issues must be validated, and the software implementation aspects must be reviewed and verified to judge the software's integrity. Lastly, runtime efficiency must be analyzed to determine if it meets expectations.

Judging the efficiency of computer software can be subjective; therefore, during the development process, the AVE simulator will be benchmarked against the Intermetrics simulator to ensure complete accuracy with regards to proper simulation of VHDL. The AVE simulator will also be benchmarked against the Intermetrics simulator to provide a comparison of processing capabilities between the two. Intermetrics' simulator is an established simulator and will provide a good model against which to judge the accuracy and efficiency of the AVE simulator.

As was previously mentioned, one of the major motivating factors for developing the VIA simulator and associated tools is to provide an integrated set of UNIX-based VHDL development tools for universities and government agencies. To support the release of the simulator to these institutions, complete documentation for all areas of the simulator must be provided. This documentation will include the details of the requirements and design analysis, fully documented source code, an installation guide, a user's manual, and a programmer's manual. The documentation will give the user information to fully understand how to run the simulator, along with providing information specific to the principles behind the simulator and how it operates. This information will allow users to modify the simulator to meet their individual needs.

1.6 Assumptions

VHDL Intermediate Access (VIA) is the form taken by the VHDL source code after it has been compiled by the AVE analyzer. This form can be readily interpreted by the simulator for the purpose of modeling a circuit described with VHDL source code. VIA facilitates the efficient execution of the simulator; therefore, the simulator is restricted to using VIA and cannot simulate a circuit directly from a description made with VHDL source code. This dependency on VIA makes implementation and testing of the simulator with each VHDL feature dependent upon the successful development and implementation of the analyzer for the same feature.

1.7 Standards

One of the major objectives of this thesis is to provide simulation facilities for VHDL as the language is defined under the new IEEE Standard 1076-1987. This standard was published in December 1987 and was unavailable for use by Maj Lynch and Lt Kodama in their work on the simulator. Now that this standard is available, the work accomplished by Maj Lynch and Lt Kodama will have to be reviewed and changes made where features do not conform to the standard.

Until a formal standard for the C programming language is adopted, de facto standards will be used to ensure that the simulator will run under any C programming environment. Presently, an ANSI standard for the C programming language is being developed; therefore, the direction provided by the draft of this standard will be the guide for developing the C source code for the simulator.

1.8 Thesis Overview

Chapter I has given the background, motivation, and direction for this thesis. The problem and its scope has been defined along with a overview of the approach to the solution.

Chapter II looks at HDLs, their history and impact and how VHDL was created to fill a void among existing HDLs.

Chapter III looks at VHDL and its ability to properly model hardware. Language constructs and semantics are compared to those desirable in an HDL.

Chapter IV provides a problem analysis and system design for the simulator.

Chapter V details the design and implementation of the simulator. Elements such as modeling code generation and simulation control are discussed.

Chapter VI reviews the design and implementation of the simulator in regards to the original design and requirements.

Chapter VII summarizes the work and provides conclusions and recommendations for further research with VHDL simulation and the AFIT VHDL Environment.

II. Hardware Description Languages and VHDL

2.1 Hardware Description Languages

2.1.1 Background. A hardware description language (HDL) is a special computer language used to describe digital circuit designs. HDLs provide a detailed, yet concise description of the design and serve as a means for engineers and circuit designers to convey their ideas to each other [Chu74]. Similar to programming languages, HDLs have a formal structure and syntax which helps to reduce ambiguity while detailing design specifics. Before HDLs were used, most designs were accomplished with pencil and paper. Diagramming on paper is fine for simple designs, but as designs become complex, the details can become confusing.

Early HDLs were typically developed as needed [Lipo77]. Most early HDLs were created for a specific application and were not general enough for another design engineer to use in a separate design. This forced circuit designers to develop HDLs to document their work rather than try to use one of the existing languages. The abundant use of application specific HDLs made it difficult for engineers to understand another's design. Unique HDLs also made it difficult to integrate two separate designs that used different languages. Some of these difficulties were eventually offset by the advent of general purpose HDLs.

2.1.2 HDLs and Simulation. Probably the biggest advantage to using HDLs over designing on paper is their suitability for computer simulation [Chu74]. A design created using an HDL can be simulated and tested on a computer prior to being constructed. This added step is very beneficial in making sure that the design is correct and complete. Finding errors before circuit construction allows for quicker and easier changes to the design and can result in substantial savings in both time and cost of developing the circuit.

As time and technology progressed, many of the original HDLs were unable to provide simulations of the ever-advancing hardware technology [Dewe86]. As hardware progressed from discrete components to integrated circuits, it became necessary to develop a new HDL that would support the latest chip technology. This language must also have the ability to support yet undeveloped technology as it becomes available. This need was the driving force behind developing VHDL.

2.2 VHSIC Hardware Description Language

2.2.1 Background. VHDL was developed under Department of Defense (DoD) guidance to become a standard HDL for DoD use. The proliferation of existing HDLs made it difficult to review designs submitted by contractors. Additionally, it was difficult to integrate two designs that were developed using different HDLs. In 1981, the Institute for Defense Analysis was tasked with the job of analyzing the requirements for a standard HDL. Part of their job was to survey the many existing languages and ensure that necessary features of these languages were incorporated into the new language [Aylo86]. After the language requirements were established, the VHDL program was begun in 1983 with the awarding of the contract to Intermetrics [Dewe86].

The primary need for a language such as VHDL was to enhance the ability to design and develop VHSIC technology for military systems [Dewe86]. Having a common language for the many and varied military applications addresses the issues of providing a means of communicating design specifics and supporting the integration of separate designs. VHDL is particularly suited to supporting a variety of applications because it is not restricted to any particular hardware technology or design methodology [Lips86]. Additionally, VHDL was designed to be intuitive in nature and suitability for simulation [Lips86]. Hardware designers find it easy to work with VHDL because the language supplies a natural methodology for modeling hardware technology.

Although VHDL was originally intended for use in DoD applications, other non-DoD institutions acknowledged its potential and began using it also. These institutions recognized the need for a standard HDL and saw VHDL as one that can meet the requirements for standardization. The motivation for VHDL "came from the Department of Defense, but the need is universal" [Waxm86].

2.2.2 The Language. VHDL allows the designer to use varying levels of abstraction in describing a design [Lips86]. At the highest level, the designer can describe a design as a function of the overall system. At the lowest level, a design can be described as a compilation of individual gates. Due to the generic nature of its language semantics, VHDL can even support descriptions below the gate level; this capability, however, was not required as part of the language requirements [Shiv85].

2.2.2.1 Design Entity At the highest level of a hardware component description is the *design entity*. The design entity constitutes a hardware component as a whole. It can be used to model a simple logic gate, complex computer system, or a level of hardware abstraction somewhere in between [Lips86]. A design entity can be comprised of other design entities. This hierarchical approach allows systems to be designed using existing components. Common components, such as simple logic gates, can be created and copied by other engineers for use in separate designs. This is analogous to building a circuit from existing off-the-shelf components.

The design entity is divided into two parts: an *entity declaration* and an *architecture body*. The entity declaration describes how the hardware component interfaces with other components external to itself. A entity declaration can be used to define a class of components, all of who's interfaces are the same but who's internal implementation may be different. In support of this component classification, the entity declaration is also used to define declarations and statements that can be shared by any design entities that may come under this class. Figure 2-1 shows the entity declaration for a full adder. The **port** list defines the input and output interface elements of the full adder.

The architecture body is the portion of the design entity that describes organization and/or operation and provides a "view" of the component. Just as hardware designs can take different approaches to implementation, VHDL descriptions can be expressed in more than one way. The different views of the same component may reflect different levels of abstraction, different types of implementations, or different styles of description [Int85b]. Three basic views provided by VHDL are *structural*, *dataflow*, and *behavioral*.

```
entity full_adder is
     port( x, y : in BIT;
               cin : in BIT;
               sum : out BIT;
               cout : out BIT)
end full_adder;
```

Figure 2-1. Entity Declaration for a Full Adder

A structural view describes the physical structure of the design and is analogous to mapping a circuit using a schematic. It describes the individual elements and how they interconnect. Figure 2-2 shows a VHDL description that gives a structural view of a full adder. In this example, lower level design entities such as the *and*, *xor*, and *or* gates are used to describe the full adder. Instantiating these components to help describe the full adder is an example of the hierarchical approach to hardware description that was discussed earlier.

The dataflow view uses signal assignment statements to show component connections and signal activity internal to the component. Figure 2-3 gives a dataflow description of a full adder. In this example, signals representing intermediate results are used to help describe the operation. These intermediate signals are local to the design entity and are not visible to other design entities.

At the opposite extreme from the structural view is the behavioral view. The behavioral view is an abstract description that represents the component entirely as a function. It describes the output of a hardware component as a function of the input. Figure 2-4 shows a VHDL behavioral view of a full adder. Procedural statements are used to implement an algorithm that describes the function of the component.

The three full adder descriptions discussed represent views that are solely of one type. VHDL also allows the circuit designer to mix structural, dataflow, and dataflow

```
architecture structural_view of full_adder is

        -- component declarations
        component and_gate port(a, b : in BIT; C : out BIT);
        end component;
        component and_gate port(a, b : in BIT; C : out BIT);
        end component;
        component and_gate port(a, b : in BIT; C : out BIT);
        end component;

        -- configuration specifications
        for x1, x2 : xor_gate use entity xor_gate(data_flow_view)
              port map(a, b, c);
        for a1, a2 : and_gate use entity and_gate(data_flow_view)
              port map(d, e, f);
        for o1, o2 : or_gate use entity or_gate(data_flow_view)
              port map(g, h, i);

        signal s1, s2, s3 : BIT;          -- local signal declarations

        begin
              x1 : xor_gate port map(x, y, s1);
              x2 : xor_gate port map(s1, cin, sum);
              a1 : and_gate port map(cin, s1, s2);
              a2 : and_gate port map(x, y, s3);
              o1 : or_gate port map(s2, s3, cout);
end structural_view;
```

Figure 2-2. Structural View of Full Adder

statements to form a hybrid view. This flexibility allows the design engineer to develop a description that will more accurately meet the design specifications.

The same flexibility that VHDL offers the design engineer is what makes VHDL well suited for simulation. VHDL allows the engineer to isolate and concentrate on particular aspects of a design such as component functions, signals, or interfaces for verification and validation via computer simulation. Since VHDL supports behavioral descriptions at all levels of the design, a component can be analyzed at a high level of operation without being burdened with lower level details. Freeing high level descriptions from the lower levels allows for improved simulation capabilities [Aylo86].

```
architecture data_flow_view of full_adder is
     signal s1, s2 : BIT;
     begin
          s1  <= x xor y;
          s2  <= s1 and cin;
          sum <= s1 xor cin;
          cout <= s2 or (x and y);
end data_flow_view;
```

Figure 2-3. DataFlow View of Full Adder

```
architecture behavioral_view of full_adder is
   begin
     process(x, y, cin)
          variable s : BIT_VECTOR(1 to 3) := x & y & cin;
          variable num : INTEGER range 0 to 3 := 0;
     begin
          for I in 1 to 3 loop
               if s(I) = '1' then
                    num := num + 1;
               end if;
          end loop;
          case num is
               when 0 =>
                    cout <= '0';
                    sum <= '0';
               when 1 =>
                    cout <= '0';
                    sum <= '1';
               when 2 =>
                    cout <= '1';
                    sum <= '0';
               when 3 =>
                    cout <= '1';
                    sum <= '1';
          end case;
     end process;
end behavioral_view;
```

Figure 2-4. Behavioral View of Full Adder

III. VHDL and Hardware Modeling

3.1 Introduction

VHDL was developed with the purpose of achieving "faithful semantic modeling of hardware" [Lips85]. It serves to describe designs from the gate level up through the system level with flexibility to support both a behavioral or structural view [Hine87]. VHDL's strength lies in a full set of language constructs from which to model hardware characteristics along with well-defined semantics to guide model behavior. This chapter looks at VHDL in terms of its ability to model hardware by examining VHDL's constructs and semantics.

3.2 VHDL Language Constructs

3.2.1 Introduction. VHDL offers the circuit designer many language constructs to accurately model a digital system. The language supports data abstractions, process control, and expressions that parallel hardware designs. At the heart of VHDL is the signal assignment statement from which the designer can produce a structural, behavioral, or dataflow description.

Based on the Ada programming language, VHDL inherits much of Ada's structural and procedural aspects. The ability to describe hardware characteristics algorithmically gives VHDL additional power and flexibility. Users familiar with Ada or other high-order languages will recognize VHDL's procedural features. The high-order language nature of VHDL also help make it tool and machine independent.

To provide data abstraction, process control, and expressions to model hardware designs, VHDL offers a range of language constructs. These constructs are presented and discussed under language feature categories identified as being desirable for an HDL: data types, operations, statement types, operator definitions, order of execution, block structure, instantiation of components, and process structure [Barb81].

3.2.2 Data Types and Objects

3.2.2.1 Data Types. A data type is a declaration of an object, such as an integer, and a set of operations that are associated with this class of objects [IEEE88]. An HDL should support a range of data types that map well to hardware. Besides common data types such as integer, real, character, and string, an HDL should support hardware associated types such as bit, boolean, and bit vector.

VHDL is a strongly typed language and supports four classes of data types: *scalar*, *composite*, *access*, and *file* [IEEE88]. Scalar types include *integer*, *floating point*, *enumerated*, and *physical types*. Integer and floating point are numeric types and are similar to those of other programming languages. An enumerated type is an explicit list of literal values that are to be associated with a declared instance of an enumerated type. Predefined enumerated types for VHDL are **bit**, **boolean**, and **character**. Each of these types is defined by declaring all instances of the type. For example, if type **bit** was not already predefined in package standard, a user could create it by declaring:

```
type bit is ('0', '1');
```

Physical types are used to denote objects that have a physical unit of measure. The physical type declaration defines incremental measures of some base unit. Figure 3-1 shows how a physical type declaration can be used to define units of resistance. The base unit for this declaration is nOhm; other resistance measures are defined as multiples of this base unit. The underlying representation of a physical type's base unit is the integer value 1. Each incremental measure of the physical type's base unit is represented by an integer that is equal to the ratio of the incremental measure to the base unit. The range of the physical type is dependent upon the largest integer supported by the host machine. Physical types support two aspects of hardware modeling. First, they permit the user to use common terms to define physical units. Secondly, the underlying integer value representation facilitates simulation by making computations on units of physical type integer operations.

```
type resistance is range 0 to 2**31-1
    units
        nOhm;
        uOhm = 1000 nOhm;
        mOhm = 1000 uOhm;
        Ohm = 1000 mOhm;
        kOhm = 1000 Ohm;
        megOhm = 1000kOhm;
    end units;
```

Figure 3-1. Declaration of a Physical Type [CLS87a:37]

Composite types are made from a collection of one or more types. VHDL composite types include *arrays* and *records*. Composite types can be used to represent a collection of objects from the model. For example, an array of type **bit** can be used to model a bus. An array of type **bit** is actually predefined by VHDL and is called a **bit_vector**. Arrays can be used to model other objects such as registers, memory locations, or buffers.

Records are a collection of elements that must be of the same class of objects. These elements, however, may be of different types from that class. Figure 3-2 shows an example of a record that is used to describe a component description.

```
type component is
    record
        name : string(1 to 30);
        id : natural;
    end record;
```

Figure 3-2. Declaration of Type Record

Access types are comparable to pointer types in other languages. These are used to dynamically create and delete objects of type **variable**. File types represent disk files as defined under the host operating system and define the type of objects to be stored in these disk files. The user can use files to inject signal values into the model to test for various conditions. Files also let the user extract specific information about the model and store this information into a disk file for later evaluation. These options help isolate the information of interest to the user, or they can be used to help debug the design.

VHDL also supports *user-defined* types. User-defined types allow the circuit designer to create objects that map directly to hardware objects. For example, a user-defined type can be used to represent a signal that has multi-valued characteristics such as one that assumes the following states: high, low, high impedance, and undefined. Figure 3-3 shows how a multi-valued signal can be described with VHDL. This type includes an *error state* that is used to signal an error condition during simulation.

```
type multi_value_logic is
       ( 'U'                    -- undefined state
         '0',                   -- low logic
         '1'                    -- high logic
         'Z'                    -- high impedance
         'E')                   -- error condition

signal A : multi_value_logic;
```

Figure 3-3. Multi-Valued Signal

3.2.2.2 Objects. An instance of a type assumes that type's characteristics and is called an *object*. With VHDL, the user can create objects of four classes: *constants*, *variables*, *signals*, and *files*. At any given time during simulation, the values assumed by all objects in the VHDL model signify the current state of the model [CSL87a].

Constants and variables refer only to a value of an object. Constants are initialized at the start of the simulation and remain unchanged throughout the simulation. Variable objects may change value during simulation via the variable assignment statement. [CSL87a]

Signals also have an associated value which corresponds to its current state at any given point during the simulation. A signal differs from a variable by its associated signal attributes. These attributes include a history of past values and a set of projected future values. A signal also inherits attributes associated with digital signals.

Attributes are not specific to just signals. The attribute construct "is a value, function, type, range, signal, or constant that may be associated with one or more entities in a description" [IEEE88:4-14]. VHDL supports two types of attributes: *predefined* and *user-defined*. Predefined attributes denote characteristics of the model such as high and low bound values of an object. User-defined attributes provide model characteristics also, but they are explicit to the user's needs. One application for user-defined attributes is to permit designers to extract circuit information that can be used to drive various CAD tools [Lips86].

3.2.3 Primitive Operations. The definition for a data type implies that objects of a specific type inherit certain operations. Some operations reflect the procedural aspect of the language while others provide mechanisms to describe hardware behavior. Classes of primitive operations for an HDL should include *logical*, *arithmetic*, and *relational* functions [Barb81]. Logical operations include such functions as *and*, *or*, and *not* that can be used to model a network of logic gates. Arithmetic operations such as addition, subtraction, multiplication, and division operate on numeric and some physical types. These operations permit functional descriptions that utilize procedural statements. Relational operations permit comparison between objects of the model and also support functional descriptions.

The VHDL Language Reference Manual [LRM] defines an expression as a "formula that defines the computation of a value" [IEEE88:7-1]. The expression's operand types determine the type of value being computed. VHDL supports a full set of primitive operators for its pre-defined types. Table 3-1 lists the classes of primitive operators and their associated operations. Provisions for logical, arithmetic, and relation operations are present. The LRM details the rules and priorities of expression evaluation for the different classes of operations [IEEE88:7-2].

Table 3-1. VHDL Operators

Logical	and	or	nand	nor	xor	
Relational	=	/=	<	<=	>	>=
Adding	+	-	&			
Sign	+	-				
Multiplying	*	/	mod	rem		
Miscellaneous	**	abs	not			

3.2.4 Statement Types. For an HDL, a statement should "describe both the operator of the system and the connection of components" [Barb81]. There are three types of statements: *data*, *control*, and *declarative*. Data statements cover the processing of data; control statements control the ordering and execution of operations; declarative statements define the structure and interfaces of the system. [Barb81]

Signal and variable assignment statements are the means by which values are passed among objects of a VHDL model. These are primitive VHDL statements upon which circuit descriptions are built. A more detailed look at the assignment statement is given in section 3.3.2.

VHDL offers a set of control structures that permit repetitious, selective, or alternative execution of statements. Conditional statements such as the **if-then-else**, and **case** statements are mechanisms to provide alternative processing based upon current circuit parameters. The **loop** construct offers the ability to repeat operations based on circuit conditions or for a range of values. The selected and guarded signal assignment statements bring conditional control to the primitive level of description. Execution control statements such as the **wait**, **next**, and **exit** provide explicit user control for process execution.

The declarative statement is used in VHDL to define various entities used to model hardware. The forms of the declaration include *type*, *subtype*. *object*, *integer*, *component*, *entity*, and *package* declarations [IEEE88]. Each declaration associates an identifier with the entity type of the declaration. Some declarations in VHDL provide for an initialization clause. This permits the user to specify the value an object is to assume at the start of simulation.

3.2.5 Operator Definitions. To create the behavioral description of a system, it may be necessary to represent operations in more abstract terms than can be described with primitive operators. User defined operations, functions or macros permit the description of hardware at levels that can accurately characterize behavior without being burdened with individual primitive operations [Barb81]. The abstract operations also permit the user to describe unique or complex operations of a system that cannot be modeled with primitive operators. VHDL offers two methods for creating user-defined operators: *overloading* and *resolution functions*.

3.2.5.1 Overloading. VHDL's primitive operators are defined only for its predefined types. For example, the **and** operation is defined only for types **bit** and **boolean.** This means that the primitive **and** operator may not be use for user-defined types such as that of Figure 3-3. Instead, circuit designers may define a new **and** operator to be used for their new multi-valued type. This redefinition of an existing operator is called *overloading.* Figure 3-4 shows how the **and** operation can be *overloaded* for the multi-value type that was defined in Figure 3-3. Note that overloaded operations may be used in either infix or prefix notation.

3.2.5.2 Resolution Functions. The resolution function determines the resultant value for a signal that is being driven by more than one source. One application for a resolution function is to determine the value of the signal on a data bus that is being driven by more than one source. Whenever a source asserts a value, the resolution function must take into consideration all other sources before determining the resultant value for the bus. The resolution function can be used to detect conditions of error and report such instances. Signals that are to be resolved must have the resolution function specified in their declaration.

```
function and(A, B : multi_value_logic) return multi_value_logic is
        signal temp_signal : multi_value_logic;
    begin
        temp_signal <= 'U' when (A = 'U') or (B = 'U');
        temp_signal <= '0' when (A = '0') or (B = '0');
        temp_signal <= '1' when (A = '1') and (B = '1');
        temp_signal <= 'U' when (A = 'Z') or (B = 'Z');
        temp_signal <= 'E' when (A = 'E') or (B = 'E');
        return temp_signal
end
           •
           •
           •
signal A, B, C, D : multi_valu_logic;

C <= and(A,B);
D <= A and B;
```

Figure 3-4. Example of Overloading

3.2.6 Order of Execution. Support for both concurrent and sequential operations are necessary to properly model most digital hardware devices. The fact that a language is procedural, nonprocedural, or a combination of both directly impacts its ability to support concurrent and sequential operations. In addition, control over statement execution should be available.

VHDL provides both procedural and nonprocedural constructs. Concurrent VHDL statements such as the concurrent signal assignment statement imply no order of execution and correctly model independent behavior of a circuit. For sequential characteristics, the process statement may be used. The process statement allows the use of the full set of VHDL procedural constructs.

3.2.7 Block Structure. VHDL is a hierarchical language with various constructs to support modular designs. At the lowest levels of description, VHDL uses the **block** statement to group common statements or organize the structure of the design. At a higher level, the design entity concept facilitates a modular organization of the design.

The design entity usually represents a component and may itself be composed of other design entities.

VHDL borrowed the *package* language concept from Ada to provide a means to group similar portions of the design [Nash86]. Additionally, packages can be used to isolate technological dependencies of the design [Nash86]. Figure 3-5 shows how circuit attributes can be grouped in a separate package to facilitate changes necessary to model under different hardware conditions.

```
package hardware_attributes is
     constant word_size;           : integer := 32;
     constant data_bus_width;      : integer := 16;
     constant address_bus_width    : integer := 32;
end hardware_attributes;
```

Figure 3-5. Package

3.2.8 Instantiation of Components. Modeling hardware often correlates to the actual building of the components where a building block approach is taken. Many of the components used in digital hardware are based upon common components such as gates, registers, etc. Just as the hardware engineer takes parts off the shelf to build a component, VHDL components can be taken from a common design library and included in various designs. The component instantiation statement permits the user to include pre-existing design entities. VHDL supports a multi-dimensional hierarchy of instantiation. Instances of a component that are included into a design may themselves use component instantiation. Component instantiation simplifies the design process, and it reduces design error by giving the circuit designer a library of fully tested design entities from which to draw.

3.2.9 Process Structure.

The instance of a process in hardware usually refers to the behavioral aspects of discrete portions of the component. Although these processes usually operate concurrently, often they must interface with one another to pass values or to synchronize their operation. It is important that an HDL be able to model such activity.

The basic unit of action of the VHDL model is the process [Bart88]. The signal assignment statement and the process statement are examples of this process model. The signal assignment represents discrete signal activity within the hardware while the process statement gives the designer the tool to describe behavior in a more abstract manner.

The coding of a process using the process statement is similar to a coding a procedure with a typical high order language such as Ada or Pascal. The statements within a process statement are executed sequentially with procedural type statements such as conditional and iterative statements used to help describe the behavior. Figure 3-6 shows a conditional signal assignment statement and its equivalent process statement.

```
S <= "01" when A < B else
     "10" when A > B else
     "00";
```

```
process(A,B)
begin
    if A < B then
        S <= "01";
    elsif A > B then
        S <= "10"
    else
        S <= "00";
    end if;
end process;
```

Figure 3-6. Signal Assignment Statement Versus Process Statement [Lips86:30]

3.3 VHDL Timing Concepts

3.3.1 Introduction. An HDL should support accurate description of timing characteristics when describing a hardware model [Aylo86]. Most HDLs serve to support simulation of digital circuits, but often the issue of hardware timing is left to the simulator. VHDL, on the other hand, has well defined timing semantics inherent in the language [Hine87]. These timing semantics include specific language features to give the circuit designer the power to truly describe the circuit operation plus a well defined model which the language must follow to simulate true hardware operation.

This section analyzes VHDL with respect to its ability to model the various timing characteristics found in digital circuitry. Since much of the timing aspects of VHDL are tied to the signal assignment statement, their role will be discussed first. The signal assignment statement is the primitive expression upon which all descriptions are built. They allow the designer to describe a full range of circuit characteristics and control the basic timing aspects of the circuit.

3.3.2 Signal Assignment Statements

3.3.2.1 General. VHDL uses an expression called the *signal assignment statement* to assign a value to a signal. An example of a signal assignment statement that represents an AND gate is:

A <= B **and** C;

There are three types of signal assignment statements supported by VHDL: *timed*, *conditional*, and *guarded* [Bart88].

3.3.2.2 Timed Signal Assignment Statements. To properly model digital hardware, an HDL must have the ability to specify timing characteristics of the components. One such characteristic is the delay associated with the transfer of input signals to an output signal. Again using an AND gate, VHDL can describe the same expression as above but for an AND gate that requires 10 nanoseconds (ns) to propagate the signal:

A <= B and C after 10 ns;

This 10 ns delay is taken from the time the statement is executed. The delay may also be specified as a variable or function. This permits the modeling of hardware that has a variable delay or a delay that is dependent upon other factors of the circuit. Figure 3-7 shows how VHDL can model a component with a variable delay. This example uses the function *gate_delay* to compute the delay based on temperature conditions at the time of execution.

```
function gate_delay(temp : temperature) return time is
    begin
        if temp < 0
            return(10ns);
        else
            return(15ns);
    end;

A <= B and C after gate_delay;
```

Figure 3-7. Variable Timing Delay

3.3.2.3 Conditional Signal Assignment Statements. There are two signal assignment statements that assign values to signals based upon circuit conditions [Bart88]. The first is the conditional signal assignment statement. This statement lists a sequence of expressions that may be used to determine the assigned value for the signal. The conditions are tested in order of presentation until one is found to be true. The expression associated with this true condition is then executed. Figure 3-8 shows a description of an R-S flip-flop using a conditional assignment statement. In this example, UNDEFINED is a user defined function used to determine the value under this undefined condition.

```
signal q, r, s : BIT;

     q <= q when (r = '0') and (s = '0') else
     q <= '1' when (r = '1') and (s = '0') else
     q <= '0' when (r = '0') and (s = '1') else
     UNDEFINED;
```

Figure 3-8. R-S Flip Flop [Bart87:42]

The second conditional signal assignment statement is the selected signal assignment statement. This statement lists values for the expression along with conditions to be met for each value to be assigned to the signal in a manner much like a case statement. Figure 3-9 shows a description of an eight- input inverted multiplexer using the selected signal assignment statement.

```
signal d0, d1, d2, d3, d4, d5, d6, d7, a, b, c, w : BIT;

with (a & b & c) select
     w <= not d0 when "000",
     w <= not d1 when "001",
     w <= not d2 when "010",
     w <= not d3 when "011",
     w <= not d4 when "100",
     w <= not d5 when "101",
     w <= not d6 when "110",
     w <= not d7 when others;
```

Figure 3-9. Eight-Input Inverted Multiplexer [Bart87:41]

3.3.2.4 Guarded Signal Assignment Statements.

For synchronous circuits, signal behavior is controlled by a clock [Bart88]. The same clock state may affect a group of common signals. VHDL permits the grouping of related signals using the **block** statement. Each **block** statement may have an optional *guard* expression that must evaluate to true for the guarded signal assignment statements contained within the block to be evaluated. The guard expression is enclosed in parenthesis and is placed immediately after the **block** statement. Figure 3-10 shows a VHDL description of a D flip-flop that utilizes a guard expression. The keyword **guarded** is used with expressions that are to be evaluated only when the guard expression is true. For the D flip-flop of Figure 3-10, the input signal is clocked to the output only when CLK is high.

```
signal CLK, q, qb, d : BIT;

flipflop: block(CLK = '1')
     signal s : BIT;
     begin
          s  <= guarded d;
          q  <= s;
          qb <= not s;
end block flipflop;
```

Figure 3-10. D Flip-Flop [Bart87:42]

3.3.2.5 Waveforms.

A *waveform* is represented by the expression on the right hand side of a signal assignment statement [CLS87a]. This waveform reflects values being asserted by a signal's driver (see next section). Each waveform may contain one or more waveform elements with each waveform element specifying a signal value and the delay after which the signal value is to be assigned. Figure 3-11 shows a signal assignment statement with multiple waveform elements for an integer, A:

```
signal A : integer;

A <= 1 after 10 ns, 2 after 20 ns, 3 after 30 ns;
```

Figure 3-11. Multiple Waveform

3.3.2.6 Drivers. Each signal has one or more *drivers* which contain its current value and associated waveform. Again, each waveform element contains a value and the time at which the signal assumes this value; therefore, a driver not only lists the current signal value, but also all values it will assume in the future. Signals with multiple drivers must have their value resolved from those being asserted by all drivers (see sect 3.2.5.2). Figure 3-12 pictures a driver for signal A of Figure 3-11 and shows how future values are queued to be resolved when their projected future time arrives. Waveforms and drivers are discussed further in section 3.4.3.2.

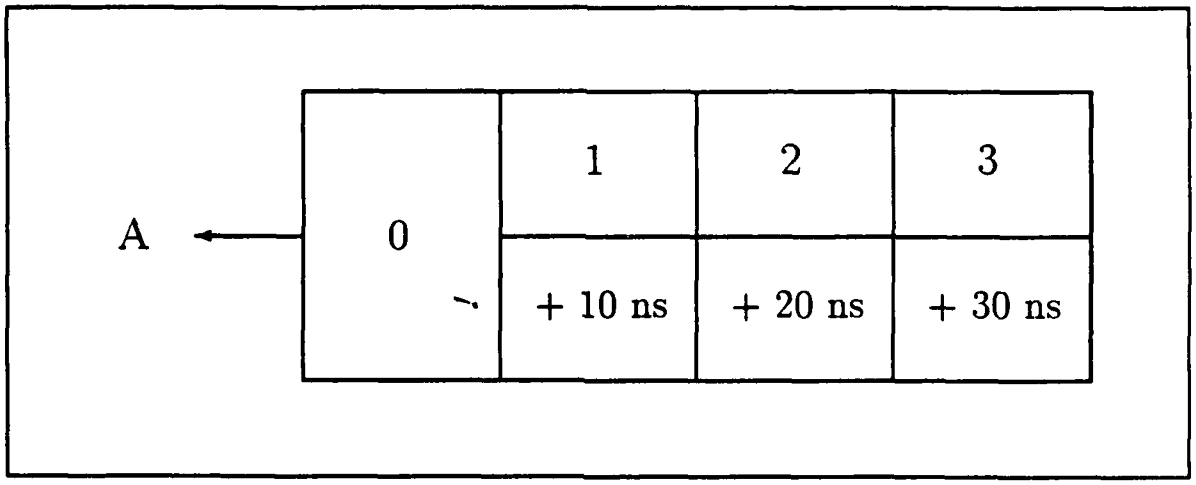

Figure 3-12. Picture of Driver

3.3.3 Delay

3.3.3.1 General. There are two time delay models supported by VHDL: *inertial* and *transport* [IEEE88]. These two models help the circuit designer to accurately model digital hardware as it would actually operate.

3.3.3.2 Inertial and Transport Delay. Inertial delay represents the delay attributed to switching circuits [IEEE88]. This delay period corresponds to the time needed for a component to propagate the input signal to the output. If the input signal remains valid for a time greater than the inertial delay, then its value is propagated; otherwise, no change is made at the output. Since inertial delay is common in digital circuits, it is the default delay model in VHDL [CLS87a].

The transport delay model is optional for signal assignments and is used to model hardware that exhibits no inertial delay [IEEE88]. These devices have a nearly infinite frequency response and changes in input, no matter how short, are always reflected on output. Since this time model is optional, the keyword **transport** must be used with the signal assignment statement to indicate a transport delay.

Figure 3-13 shows two signal assignment statements, one with an inertial delay and one with a transport delay. The transients of the originating signal, S, have a shorter duration than the specified inertial delay and are ignored by the inertial model. In this example, propagation of transients signals are undesirable; therefore, the inertial model should be used.

When all changes in the input signal are significant, the transport model should be used. Figure 3-14 shows a signal, S, that represents a signal to be sent over a transmission line. This signal carries bits of information that are represented by short signal pulses. Each pulse is significant even though they they last for a period of time that is shorter than the line delay. To correctly model the transmitted signal, the transport model must be used. The resulting waveform is represented by signal B and reflects a 5ns delay for the line. If the inertial delay model was used, the resulting signal would not reflect the code groups. Signal A shows the result if the inertial delay model was used.

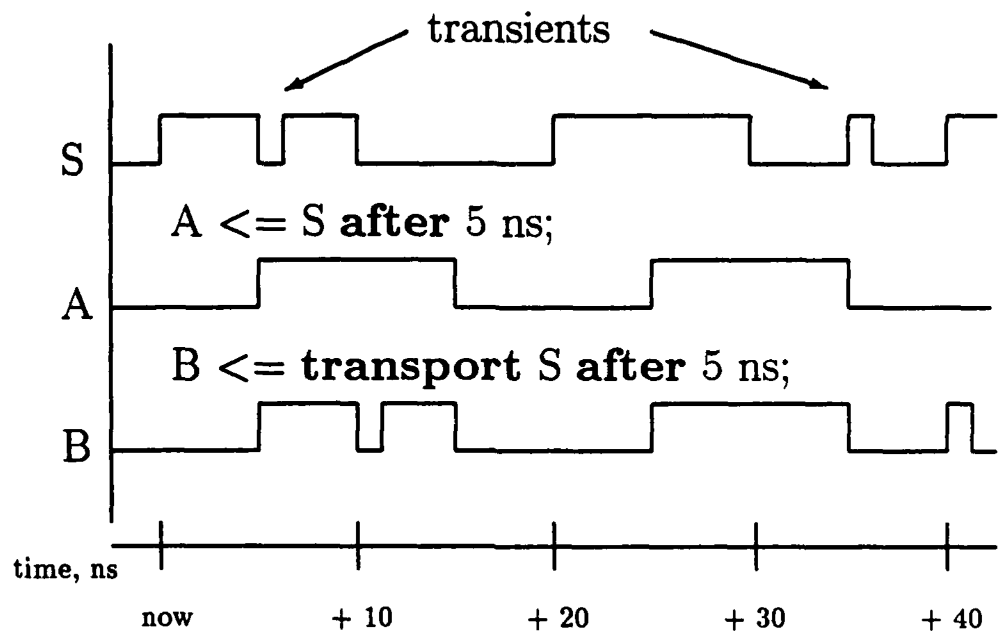

Figure 3-13. Modeling Propagation Delay Through a Buffer [CLS87a:59]

3.3.3.3 Delta Delay. When no timing clause or a zero delay timing clause is specified, VHDL assumes an infinitesimally small unit of time for the delay. This delay, called *delta delay*, is greater than zero but smaller than the smallest measurable unit of time. Any number of delta delays added together are as a whole still smaller than the smallest measurable unit of time. This delay accurately models hardware where some minimal amount of time is required for changes of state within the hardware model. This delay also gives order to sequential events that occur at the same simulation time. Figure 3-15 shows an example of VHDL code that assumes delta delay along with the signal waveforms generated for this model.

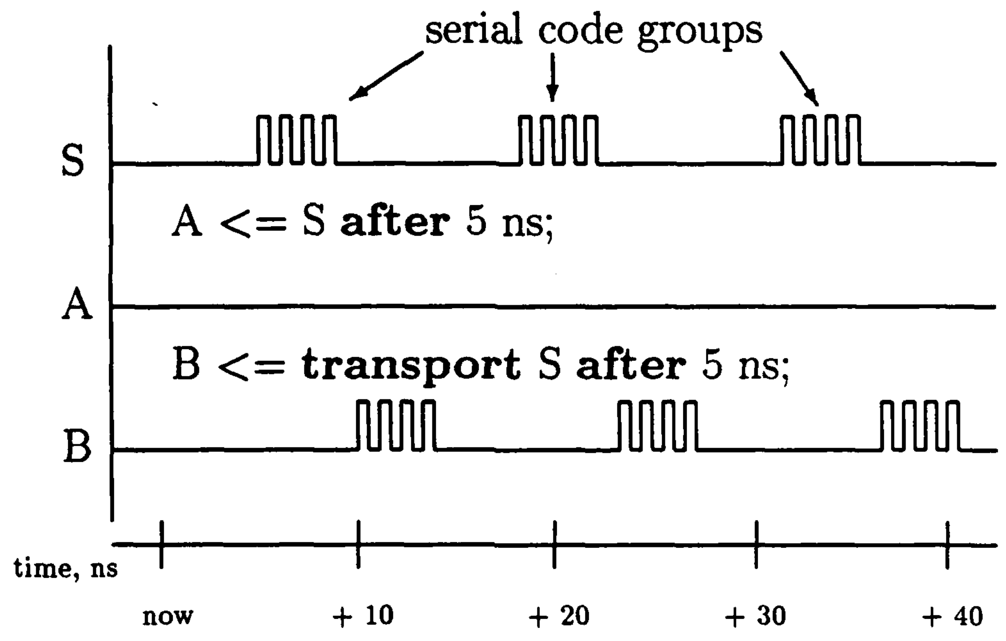

Figure 3-14. Modeling Transmission Line Delay [CLS87a:60]

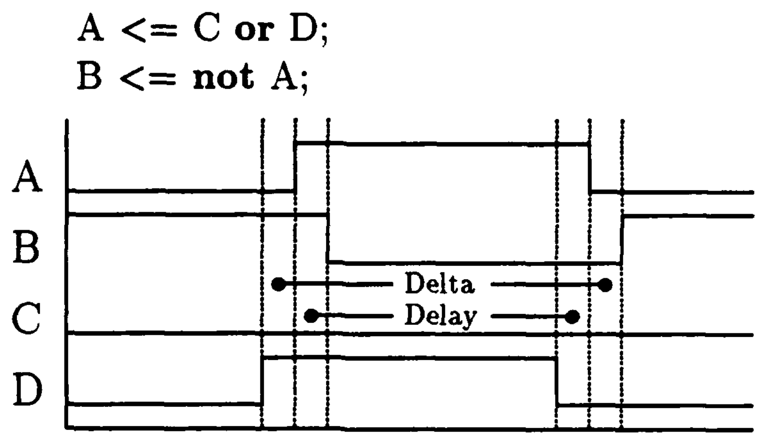

Figure 3-15. Delta Delay

3.3.4 Timing Characteristics

3.3.4.1 Constraints. Normal digital devices operate under constraints that can be attributed to hardware operation. Typically, hardware operation incurs constraints such as set-up times, hold times, and pulse width minimum/maximum values. The set-up time for a component is the length of time that a signal must be stable before a dependent signal can change state. For example, the set-up time for a D flip-flop is the length of time the D input of the flip-flop must be asserted before it can be clocked to the output. Figure 3-16 shows the set-up time for a D flip-fiop in relation to the clock and D input while Figure 3-17 shows how VHDL can model this set-up time.

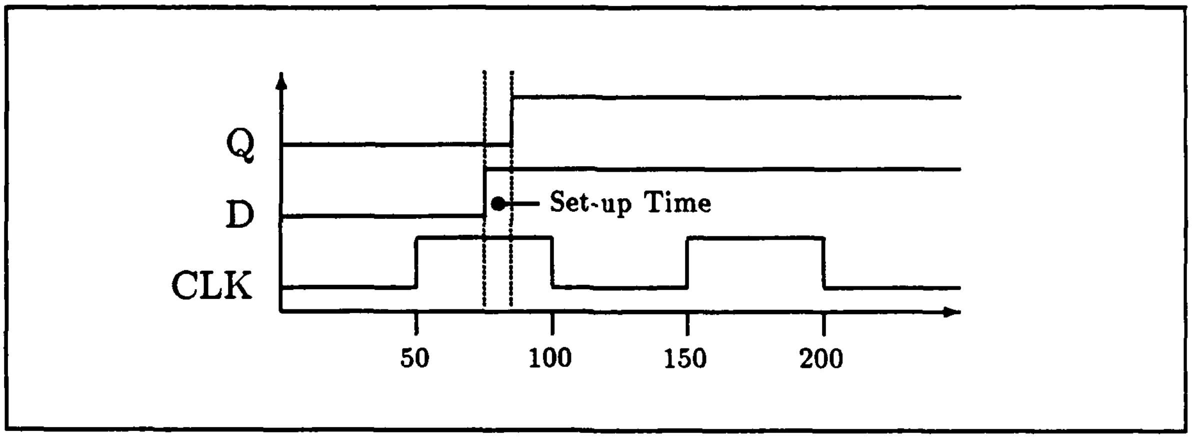

Figure 3-16. Set-Up Time

```
D_ff:block(CLK = '1')
    begin
        Q <= guarded D after 10ns;
end block D_ff;
```

Figure 3-17. VHDL Modeling Set-Up Time

The hold time for a signal is the time which a signal must be asserted after the propagation of the input signal has begun. For a D flip-flop, the hold time for the D input is the length of time D is asserted after it has been clocked. This assertion gives the flip-flop enough time to propagate the value. If the minimum hold time is not met, then an invalid output may be generated. Figure 3-18 shows hold time for the D input to a D flip-flop. Figure 3-19 gives the VHDL description that will model the hold time characteristics.

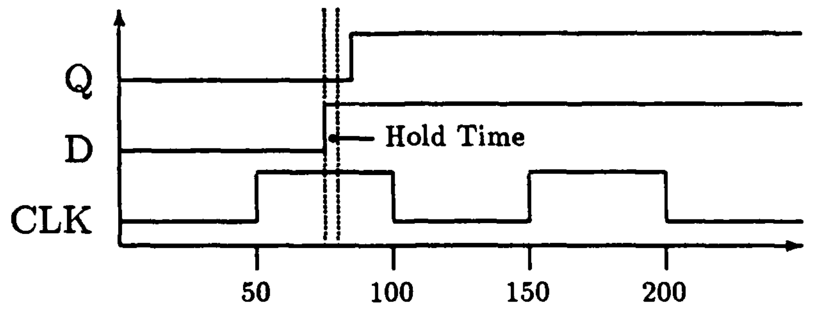

Figure 3-18. Hold Time

```
process
   begin
      wait on D;
      if(CLK = '1') then
            wait for 5ns;
            if(D'Stable(5)) then
                  Q <= D after 5ns;
            end if;
      end if;
end process
```

Figure 3-19. VHDL Modeling Hold Time

Pulse widths are often critical for proper operation of digital devices. For a transmission line, there may be a minimum pulse width that must be met for a signal to be detected. VHDL allows the design engineer to specify pulse width conditions when modeling such devices.

3.3.4.2 Rise and Fall Times. VHDL offers the ability to describe the edge characteristics of signals. In particular, the designer can model rise times, fall times, and periodicity. With VHDL, the designer can specify transition times between states with both constant delays and variable delays.

Signal transitions in a circuit do not occur instantaneously; there is some amount of time required for the signals to change states. In the case of boolean logic, the rise time is that which is needed for a signal to change from 0 to 1. The fall time is the reverse—the time needed for a signal to change from 1 to 0.

With VHDL, the designer can model both rise and fall times for hardware components. For example, the VHDL selected signal assignment statement can be used to model an inverter that has a rise time different than its fall time. Figure 3-20 gives the VHDL description of an inverter with such characteristics.

```
B <= not A after 10ns when B = '0' else
     not A after 5ns;
```

Figure 3-20. Inverter with Differing Rise and Fall Times

3.3.5 Clock Definitions. VHDL supports several methods for generating a clock. The VHDL Tutorial [CLS87a] gives examples of some of these methods. First, a delayed signal assignment may be used to generate a simple clock. Figure 3-21 shows the VHDL description for generating a clock with a specified period. The **constant** declaration establishes the period for the clock; this value will not change during the simulation. The initialize clause used with the clock declaration initializes the clock to '0' at the beginning of the simulation while the statement at label OSC inverts the clock at every half period. This code will generate a square wave pulse as shown in Figure 3-22.

If the duty cycle for a clock is other than 50% then the duration for the high portion of the clock will be different than that of the low portion. This characteristic can be modeled with VHDL using the conditional signal assignment statement as shown in Figure 3-23. ***high_time*** and *low_time* can be either constant values that were predefined or variable delays.

```
entity Period_Clock is
    port(clk : out BIT);
end Period_Clock;

architecture Simple_Osc of Period_Clock is
    constant period : Time := 100ns;
    clk : BIT := '0';
begin
    osc: clk <= not clk after period/2;
end Simple_Osc;
```

Figure 3-21. A Simple Square-Wave Clock [CLS87a:68]

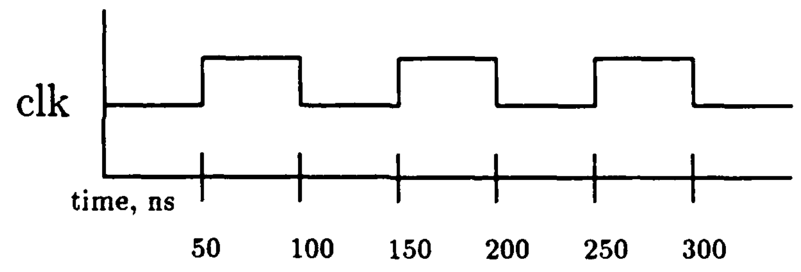

Figure 3-22. Waveform of Simple Square-Wave Clock

```
clk <= '0' after high_time when clk = '1' else
       '1' after low_time;
```

Figure 3-23. VHDL to Model Clock

For more complex clocking, a selected signal assignment statement can be used to explicitly specify the clock characteristics. For example, the clock waveform shown in Figure 3-24 can be modeled with the selected signal assignment as shown in Figure 3-25. The **wait** statement at the end of the process suspends the process for 150ns so that each signal assignment can take affect.

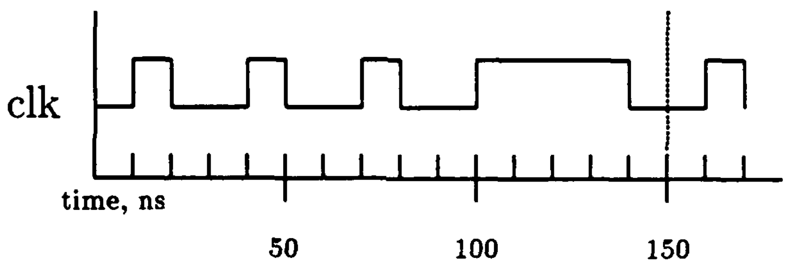

Figure 3-24. Complex Clock Waveform

Multiphase clocks can be easily modeled using a VHDL process. Figure 3-26 shows the waveform for a multiphase clock. The VHDL description for such a clock is given in Figure 3-27. Each phase is represented as a signal and their waveforms are determined by signal assignment statements.

VHDL also supports the concept of a global clock. Placing a clock definition in a separate package where it can be accessed by multiple design entities achieves this result.

```
complex_single: process
begin
     clk <= '1' after 10 ns,
            '0' after 20ns,
            '1' after 40ns,
            '0' after 50ns,
            '1' after 70ns,
            '0' after 80ns,
            '1' after 100ns,
            '0' after 140ns,
     wait for 150ns;
end process complex_single;
```

Figure 3-25. Complex Waveform Clock [CLS87a:70]

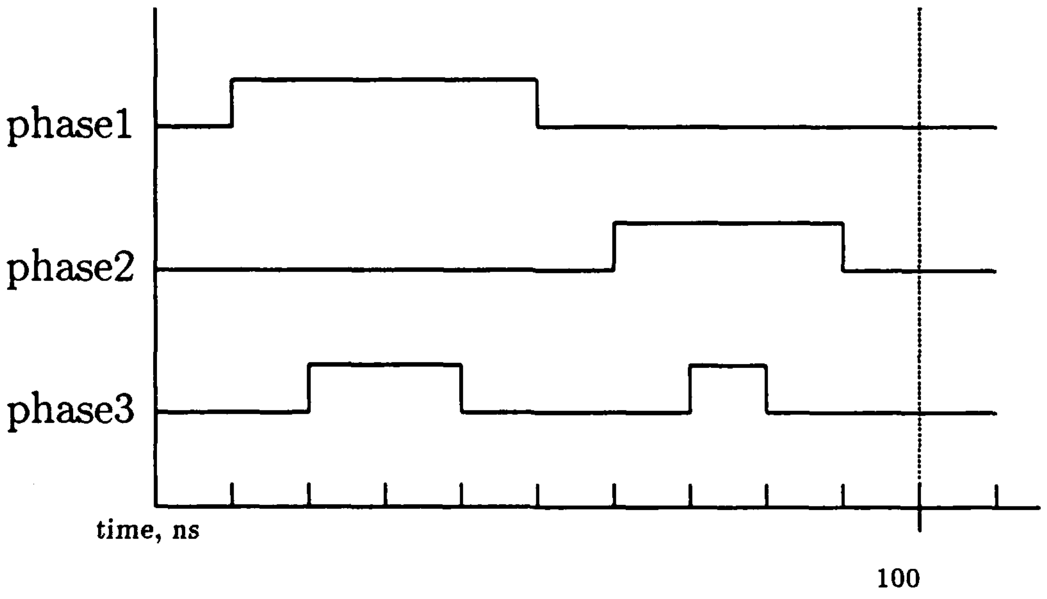

Figure 3-26. Multiphase Clock Waveform

```
three_phase: process
begin
    phase1 <= '1' after 10ns,
              '0' after 50ns;

    phase2 <= '1' after 60ns,
              '0' after 90ns;

    phase3 <= '1' after 20ns,
              '0' after 30ns;
              '1' after 70ns;
              '0' after 80ns;
    wait for 100ns;
end process three_phase;
```

Figure 3-27. Multiphase Clock [CLS87a:71]

3.3.6 Modes of Operation. Digital circuitry operates in synchronous, asynchronous, or a combination of the two. Particular to synchronous operation is the use of a clock to provide synchronization. The previous subsection showed how the generation of various clock signals can be accomplished using VHDL. Attributes of the clock signal such as rising and falling edges can also be modeled. If the designer wants to model a edge-trigger D flip-flop, the edge characteristics of the clock must be determined to model the operation. Figure 3-28 shows the VHDL description for an edge trigger D flip-flop. The *guard* expression is used to control the evaluation of the signal assignment statement. The *CLK = '1'* portion of the guard expression means that the clock must be high for the expression to evaluate to true. The *not CLK'Stable* portion means that the clock signal must not be stable. Taken together, these two expressions mean that the clock signal must have just changed to a high state for the guard expression to be true. This condition corresponds to the rising edge of the clock.

The process model of VHDL assumes no order of execution; therefore, the concurrent signal assignment statement and the process statement, which are instances of the process

```
D_ff:block(not CLK'Stable and CLK = '1')
     begin
          Q <= guarded D after 10ns;
end block D_ff;
```

Figure 3-28. Edge-Triggered D Flip-Flop

model, assume no order of execution. Building a model based on these statements can be used to model asynchronous operation.

3.4 The VHDL Simulation Cycle

3.4.1 General. One of the objectives for developing VHDL is to support CAD tools. Since VHDL descriptions model digital hardware, a simulator to analyze the model becomes a natural extension. The language provides varied constructs for the designer to build a model. When it comes time to execute the model under *simulation*, some measures of control must be undertaken by the simulator to mimic the hardware's operation. This control is inherent in the language and is clearly defined by the requirements of the language.

The VHDL Tutorial summarizes the VHDL model [CLS87a:55]:

> Two important principles determine the course of simulation in VHDL and make VHDL suitable for modeling the course of events in the physical world. First, cause always follows effect; a change in a signal value causes the execution of signal assignments that effect changes in the values of their targets. These effects may in turn cause additional changes, so that a sequence of events results. Many independent sequences of events can occur simultaneously. Second, changes can be made to take effect after some delay.

The cause-effect relationship inherent in VHDL makes an event-driven simulation model the logical choice for implementation. A simulator kernel contains the simulator executive and support routines that control the simulation. The user's model is simulated by linking the model to the kernel for execution. Execution of the model is started with an initialization phase followed by repeated execution of the model's processes.

3.4.2 Initialization. The first phase of simulation is initialization of the model. This phase consists of the following steps[IEEE88:12-14]:

1. The driving value and the effective value of each explicitly declared signal is computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.
2. The value of each implicit signal of the form S'Stable(T) or S'Quiet(T) is set to True.
3. The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.
4. Each process in the model is executed until it suspends.

Before even the above initialization steps can be taken, the objects of the model must assume some initial value. For signals and variables, an initialization clause may be specified to give each object an initial value. If no initial clause is present in the object declaration, the object assumes a value equal to the *leftmost* value for this type. For example, type **bit** has a leftmost value of '0' while the rightmost value is '1'. Initializing all objects of type bit to '0' may not give a true reflection of the hardware. User-defined types that are extensions of predefined VHDL types, such as a multi-valued logic signal (see Figure 3-3), give the user the option of defining additional states that better reflect the hardware. User defined types together with overloaded operators give the user the flexibility to define a range of varied hardware designs.

3.4.3 Simulation Cycle.

3.4.3.1 General. The simulation cycle is the process of executing the user-defined model by repeated evaluation of the model's signals and propagating these values to other dependent signals. The steps of the cycle include [IEEE:12-14]:

1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes. Simulation is complete when time advances to TIME'High.
2. Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
3. Each implicit signal in the model is updated. (Events may occur on signals as a result.)

4. For each process, if P is currently sensitive to a signal S, and an event has occurred on S in this simulation cycle, then P resumes.
5. Each process that has just resumed is executed until it suspends.

3.4.3.2 Updating Explicit Signals. Each signal in the model has one or more drivers that determine its value. With each signal assignment statement comes a new projected future value for the driver of the signal involved. The future time is determined by the delay associated with the signal assignment statement. All projected future values for a driver form a waveform for that driver. When the simulation time reaches the scheduled time for one of the projected future values, the driver's signal immediately assumes the projected value. The assignment of a projected value to a signal is called a *transaction*. If this new value is different from the signal's current value, then this signal assignment is considered an *event*.

Updating a driver's projected waveform requires more than just inserting a value at the proper time. New projected values can have an impact on other projected values. This is true for components with inertial delay whose input value changes before the value can be propagated. To properly determine future values, VHDL uses a concept called *preemptive semantics* to update the projected output waveform [Luc86b]. Under this concept, a projected value will preempt existing projected values if the existing projected values are no longer valid due to the new projected value. Figure 3-29 shows an inverter where preemptive semantics are necessary to model correct circuit operation.

The inverter in the example has a delay (10ns) associated with the propagation of an input value to the output. At time 10ns, the input value goes high and creates a projected future value of '0' for the output. This projected future value is scheduled to occur at time 20ns. At time 17ns, the input value goes low. This event creates a projected future value of '1' to occur at time 27ns. Since the first input value changed before the necessary 10ns delay needed to propagate it to the output, it's projected output waveform is no longer valid. The model must recognize the first input as a transient and delete it's projected output waveform. At time 20ns, the input goes high again. This time, the input is asserted for the necessary 10ns, driving the output low at time 30ns.

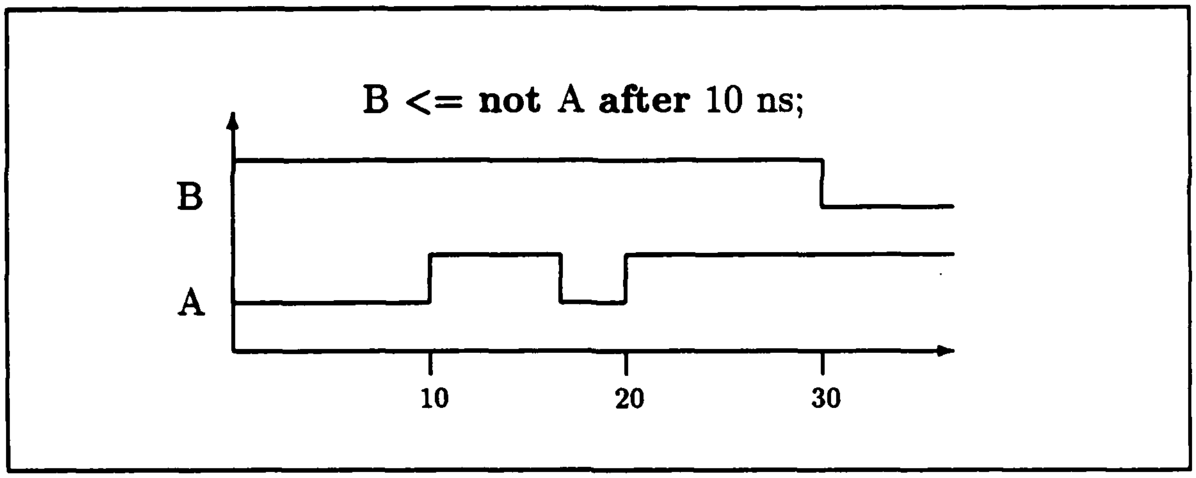

Figure 3-29. Inverter

The requirements for VHDL detail the process by which the model must follow to update the projected output waveform [IEEE88:8-5]:

1. All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform;
2. The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

If the reserved word **transport** does not appear in the corresponding signal assignment statement, then the initial delay is considered to be inertial delay, and the projected output waveform is further modified as follows:

1. All of the new transactions are marked;
2. An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;
3. The transaction that determines the current value of the driver is marked;
4. All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

If a simulator follows these steps when updating projected output waveforms, then erroneous signal values will be avoided. The first two steps keep erroneous values from being propagated when events occur that produce projected future values that are to occur before other projected future values.

The last four steps provide additional guidance to updating the projected output waveforms when the inertial delay model is used. The existence of a projected future value in the waveform means that the propagation time for this future value has not been met. Adding a new and different projected future value to the waveform means that the input value has changed before the first projected value can be propagated. This means that any different projected future values that are scheduled to occur before the new projected future value are to be preempted and must be removed from the waveform.

3.4.3.3 Updating Implicit Signals. The VHDL model assumes various implicit signals that help to define circuit characteristics. For example, the signal attribute 'Stable represents an implicit signal that has a value of TRUE when its associated signal's value has remained unchanged for a specified time. All implicit signals are re-evaluated during simulation if one of their reference signals dictates re-evaluation.

3.5 Summary

This chapter has taken a look at VHDL and the features it provides in support of hardware modeling. One of VHDL's strengths is its flexibility in supporting a range of descriptions and styles. This chapter also looked at the inherent modeling aspects of the language. VHDL's cause-effect relationship among objects along with preemptive semantics present an accurate time-based model for hardware simulation.

IV. Simulator Design Overview

4.1 System Overview

The AVE simulator is but one tool in the AFIT VHDL Environment. It allows users to simulate and verify designs that were described with VHDL. The first step of the simulation process is to have the VHDL description processed by the AVE analyzer. The analyzer parses the VHDL source code and checks for correct syntax and usage. As it parses the VHDL, the analyzer generates an intermediate form called VHDL Intermediate Access (VIA) which contains the operational characteristics of the design.

The intermediate form is formatted to facilitate usage by other development tools such as the simulator. The simulator extracts information from the VIA to build a model for simulation. This information is used to generate modeling code that is compiled and linked with the simulator kernel for execution.

4.2 Simulator Methodology

The AVE simulator is a precompiled, event-driven simulator. Previous AFIT research [Lync86, Koda87] found the number of individual elements making up a VHSIC circuit too great for all components to be efficiently simulated at the same time; however, since only 10 to 15 percent of a typical digital circuit is active at any point in time, it would be very inefficient to try to simulate all elements [dAbr85]. Event-driven simulation is a concept that uses this circuit characteristic and analyzes only the active portions of the circuit during simulation. Each signal change constitutes an event and triggers the evaluation of the changed signal and any other signals that are dependent upon this new signal value.

Another design decision for the AVE simulator concerned the use of a precompiled simulator versus an interpretive simulator. Since VHDL is for use in modeling VHSIC class circuits, the AVE simulator must be able to model circuits in this class. Due to the large size of VHSIC class designs (up to 100,000 components), simulation usually requires large amounts of memory and very long execution times, even on large minicomputers and mainframe computers. This type of resource usage can stifle productivity and have a direct impact on users, particularly in timesharing environments found at many universities. A method to reduce the resource demands of modeling VHSIC circuits was needed, and it was determined that a precompiled simulator could achieve the desired results.

A precompiled simulator helps reduce the memory requirements and long execution times inherent in VHSIC class simulation by providing more efficient execution than can be realized in an interpretive environment. Interpretive simulators are typically more efficient to set up, but they become less efficient as the number of test patterns to be simulated increases into the VHSIC realm of designs [Inte84]. Figure 4-1 shows the difference in efficiency between interpretive and precompiled simulators.

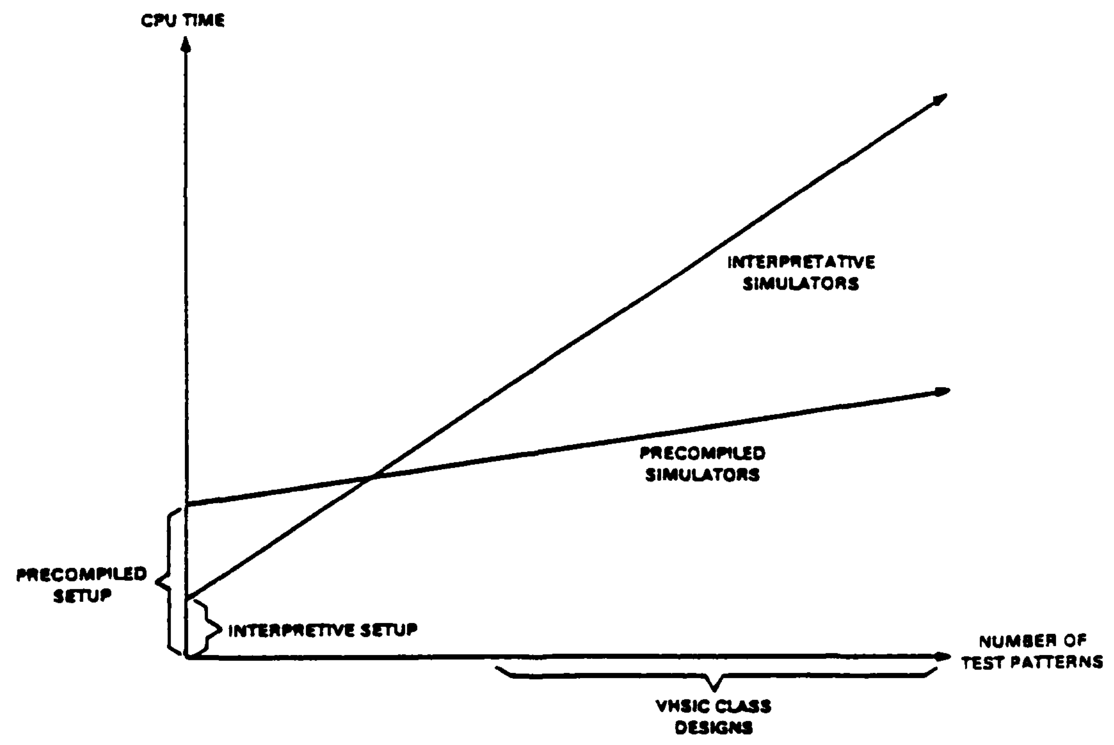

Figure 4-1. Precompilation Trade-Off Graph [Inte84:4-44]

4.3 System Requirements

4.3.1 Overview. Targeted for use by universities and government agencies, the AVE simulator must have a simple but powerful user interface to efficiently simulate realistic designs. The simulator will be developed to run under UNIX on VAX 11/785 and ELXSI computers with specific guidelines to enhance compatibility and portability to both workstation and personal computers. This section details the requirements for the simulator.

4.3.2 General Requirements. The general requirements cover those that are not directly software related. Included in this category is the support documentation for the simulator and its environment.

- A User's Manual will be supplied to detail the operation of the simulator.
- An Installation Guide will be supplied to explain how the software is to be installed.
- A Programmer's Guide will be supplied to detail the elements of the simulator and provide information to be used for maintaining and updating the software.

4.3.3 Functional Requirements. The functional requirements relate to the execution of the simulator as it pertains to proper modeiing of VHDL.

- The simulator must correctly implement the VHDL 1076-1987 timing paradigm as specified by the requirements for the language.
- Behavioral aspects of the description must be accurately modeled to reflect the description.
- Language features to be supported:
  - Types
    - Integer
    - Enumerated
      - Bit
      - Character
    - Arrays
      - Bit Vector
      - String

  - Sequential Statements
    * Wait
    * Signal Assignment
    * Variable Assignment
    * If
    * Case
    * Loop
    * Next
    * Exit
    * Null
  - Concurrent Statements
    * Block
    * Process
    * Signal Assignment
    * Component Instantiation

4.3.4 Implementation Requirements. The implementation requirements focus on the construction of the simulator software and how it will be implemented. These provide a basis from which design decisions may be made.

- The simulator will be precompiled and event-driven.
- The simulator will be implemented as two packages: Build and Simulate. The Build package will generate modeling code while the Simulate package will link the modeling code to the simulator kernel for execution.
- The simulator will operate under the UNIX operating system on a VAX 11/785, ELXSI 6400, or Sun workstation as a minimum. Operation under MS-DOS on an IBM or IBM compatible computer will be optional.
- Machine and operating system dependent portions will be isolated to enhance portability.

4.3.5 Operational Requirements. The operational requirements are those that affect the operation of the simulator from a user's standpoint. These include operational standards and user selectable options to be provided.

- The user may specify the output file to be used to store the signal transaction information.
- The user may specify the time which the model will assume at the start.

- The user may specify the model time at which the simulation will terminate.
- The user can simulate the model under specified parameters in a batch mode.
- The simulator will have the option to operate in interactive mode.
  - Setting and releasing of simulation breakpoints will be supported.
  - The user will be able to view signal values from the model when the simulation is suspended.
  - The user will be able to change signal values when the simulation is suspended.
  - The user will have the option to abort the simulation.
- The user can specify an input stimulus file used to initialize or alter object values during simulation.
- The user may specify specific signals to be traced during the simulation. Options include tracing by transaction or by event.
- On-line help will be available in both batch and interactive mode. The on-line help will be limited to listing available options.

4.3.6 System Requirements. The system requirements are those that pertain to how the simulator interfaces with its environment. Included are the interfaces between the simulator and other AVE tools and the interface between the simulator and the host machine.

- Design description input will be in VHDL Intermediate Access (VIA) format.
- Common C library functions may be used.

4.4 Simulator Design

4.4.1 Design Overview The AVE simulator is composed of two packages: Build and Simulate. The Build package generates C source code to model the design whiie the Simulate package compiles and links this modeling code with a simulator kernel for execution. Figure 4-2 shows the top-level structural analysis and design technique (SADT)[1] diagram for the system.

[1]Trademark of Softech, Inc.

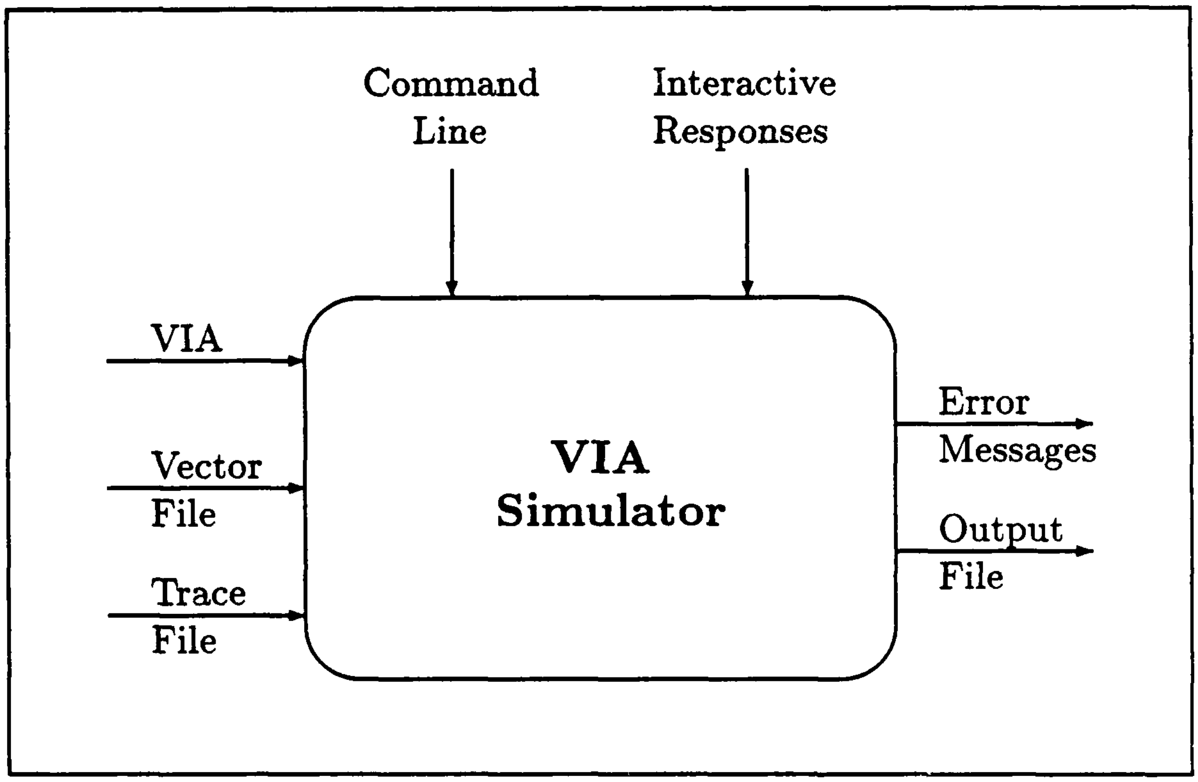

Figure 4-2. Top Level SADT View of Simulator

4.4.1.1 Build. The Build package (Figure 4-3) reads VIA and creates the data structures and modeling code for the simulation. The first step of the Build phase is to read the VIA into internal data structures for parsing. The VIA contains information in the form of a symbol table, an operation table, and a string table. The symbol table contains information about objects, such as signals and variables, used in the circuit description while the operation table contains information about the circuit operation. The symbol table also contains string literals used in the circuit description. Each of these tables is constructed by the analyzer so it can be parsed by the Build package in one pass.

4.4.1.2 Simulate. The Simulate package (Figure 4-4) begins by compiling the C source code created by the Build package and then links this modeling code to the kernel of the simulator. All circuit information translated from VIA is contained in the C modeling code and there is no need for the simulator to directly access the VIA.

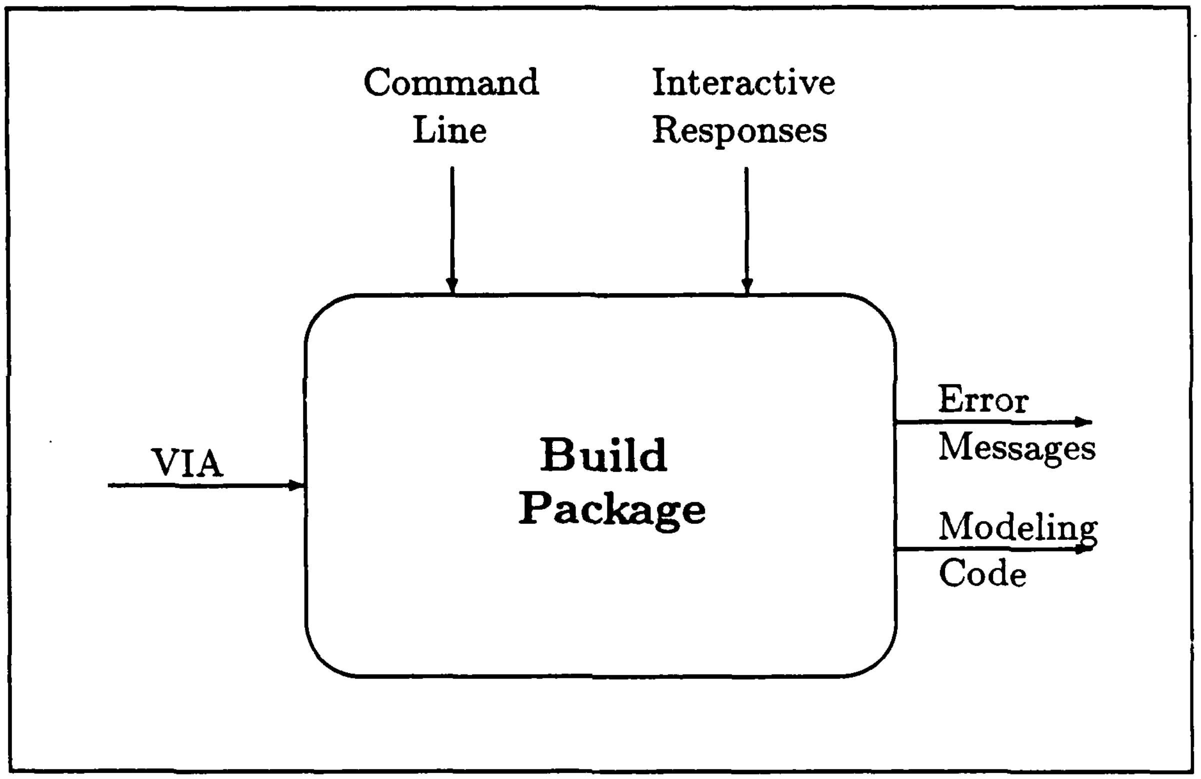

Figure 4-3. SADT View of Build Package

The simulator kernel contains contains support routines that control the simulation cycle, record circuit transactions and events, and provide control of the simulation to the user.

The central element of the VHDL model is the process. A process may be as simple as a signal assignment statement, or it may be a more abstract behavioral description using a process statement. In either case, the concept of the process is the same. All processes are independent from each other in the sense that there is no order in which they must execute; however, the execution of one process may trigger the execution of another. This is the cause-effect relationship among processes that was mentioned in section 3.4.

The cause-effect relationship among VHDL processes is depicted by processes being *sensitive* to certain signals. A process is sensitive to a signal when the signal is used to determine the value of another signal internal to the process. A change in the first signal may affect other dependent signals. The signal assignment statement

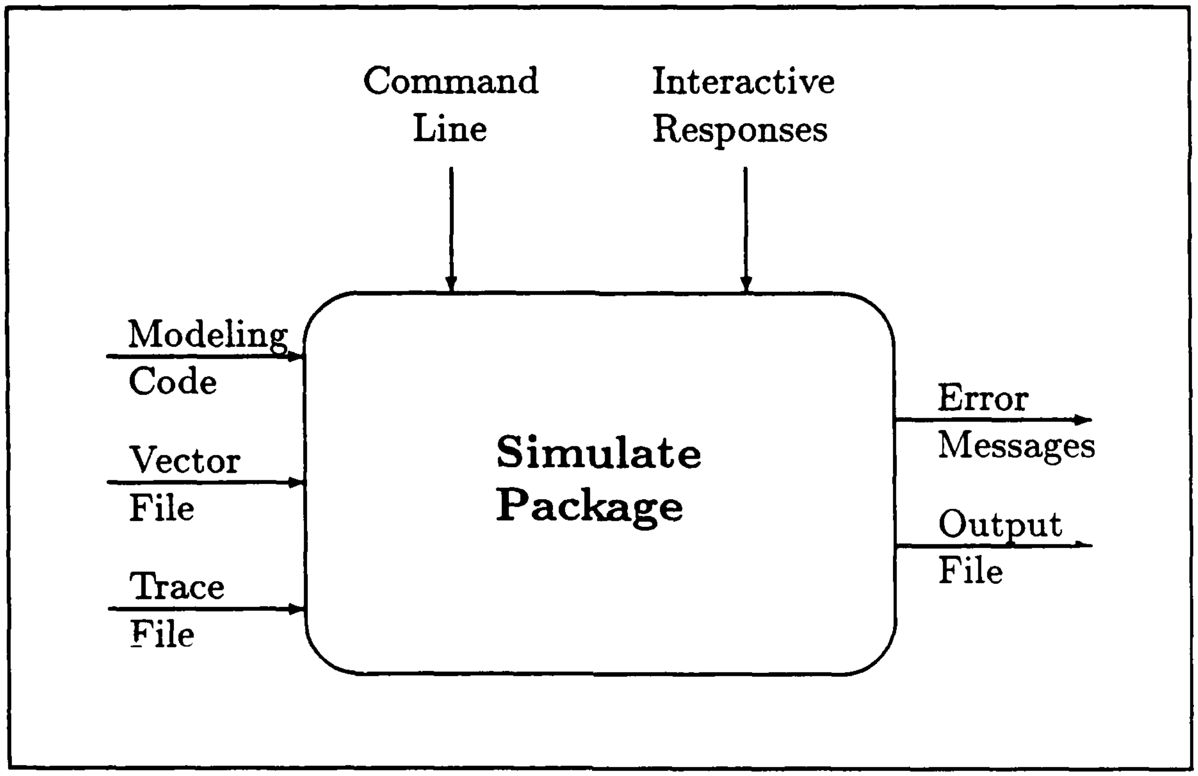

Figure 4-4. SADT View of Simulate Package

A <= B **and** C:

is a process that is sensitive to signals B and C. A change in either B or C forces a re-evaluation of the signal A. If after this re-evaluation, signal A changes value, then all processes sensitive to A must be re-evaluated also. This chain reaction mimics the propagation of signals in a circuit.

The simulation begins by executing every process in the model. When each process executes, signals are evaluated and projected future values are established for each of theses signals (see section 3.4.3.2). Later, when these signals assume their projected future values, they in turn trigger the execution of all processes sensitive to the newly changed signals.

4.4.2 Language Selection

4.4.2.1 Simulator. C was the language of choice for developing the simulator. The availability of C programming environments for a range of computers and operating systems makes this a good choice to enhance portability.

4.4.2.2 Modeling Code. Choosing the language for the modeling code must take into consideration portability of the language, its ability to link with the language of the simulator kernel, and its ability to assimilate the source language. It makes sense to use the same language for the modeling code as was used for the simulator kernel since it would simplify the linking process. C supports a full range of language constructs, most of which are comparable to the procedural constructs of VHDL. C offers flexibility in the manner in which it can be implemented. It is not strongly typed, making data conversions relatively simple. C provides support for a range of data abstractions from bit level to complex user-defined data structures.

4.5 Summary

This chapter has presented an overview of the AVE simulator. The methodology to be followed was discussed, giving some background into why a precompiled, event-driven simulator was chosen as the type to be implemented. The requirements for the simulator were listed and a design overview was given. The next chapter will continue the discussion about the simulator as it moves into implementation details.

V. Simulator Implementation

5.1 Introduction

The previous chapter discussed the requirements and principles for implementation of the AVE simulator. This chapter follows with a more detailed look at the elements of the simulator and their role in the execution of the model.

5.2 Pre-Processing

Before the AVE simulator can begin execution, the VHDL description to be modeled must be run through the AVE analyzer. The analyzer checks for syntax and static semantic errors. If no errors are found, the analyzer creates an intermediate form that represents the model. This intermediate form, called VHDL Intermediate Access (VIA), is a series of records that contain information about the original design. The VIA is stored as an ascii text file with one record per line. An intermediate form such as VIA provides a representation of the design that can be readily used by all AVE tools.

The VIA is composed of four parts: the header, the symbol table, the operation table, and the string table. The header lists the number of records in the symbol and operation table and the number of characters in the string table. The symbol and string tables translate into the objects of the model while the operation table parses into the modeling code. A detailed explanation of the VIA can be found in Berk's work with the AVE analyzer [Berk88].

5.3 Build

The purpose of the Build phase is to generate the modeling code for simulation. The steps of the build process are as follows:

1. Read VIA File
2. Parse Symbol Table
3. Parse Operation Table

5.3.1 Read VIA File. The first step of the build phase is to read the VIA from disk file. The entire VIA description is read into memory for the sake of processing speed. Each record of the VIA is read into an internal data structure, one for each of the three VIA record types: symbol table record, operation table record, and string table record. The fact that the VIA operation table is parsed in one pass would allow the input routines to be modified to read the information as it is processed. This modification may be necessary for systems where available memory restricts from reading the entire VIA description into internal structures.

5.3.2 Parse Symbol Table. Once the VIA is read, processing begins by parsing the symbol table to establish the objects of the model. A VHDL model's constants, variables, signals, and files are considered its objects. For those objects which assume dynamic characteristics, the simulator must create a variable that assumes that object's attributes and characteristics during the simulation.

Since there are four types of objects, and each of these objects may be one of several different types, the simulator must generate a variable that is comparable to the model's original object. For example, if the model has a variable that is of type integer, the simulator will generate an integer variable to represent this object. Specific properties of the object, such as valid range of values, must be controlled by runtime checks performed by the simulator.

Constants are values declared by the user in the VHDL source code. Since these objects have no dynamic characteristics to be modeled, the analyzer "hard codes" their value at every instance of their use. No data structure is needed by the simulator to maintain these objects.

Generation of objects becomes more difficult as the objects assume more complex data types. If the object in the above example was a variable that was an array of integers, then the model of the object must reflect the nature of the array. A more detailed explanation of how the simulator models VHDL data structures is given later in this chapter.

5.3.2.1 Signals. Signals are the most complex of the objects. Not only do signals assume a current value, they also assume a history of past values and a set of projected future values. The data structure used for a signal must either contain all relevant information or have links to where this information can be found. Figure 5-1 shows the representation used for a signal. Stored within the signal structure are current value, last value, time of last event, time of last transaction, size, element, and low bound. Current and last values reflect the signal's current and previous states. Time of last event and last transaction store the simulation times of these activities. Size, element, and low bound are used for composite structures and refer to the size of the structure, what element this signal occupies in this structure, and the first element of the structure. The signal structure also contains links to other related structures. These include links to structures that represent the signal drivers along with a link to an array of processes that are sensitive to this signal.

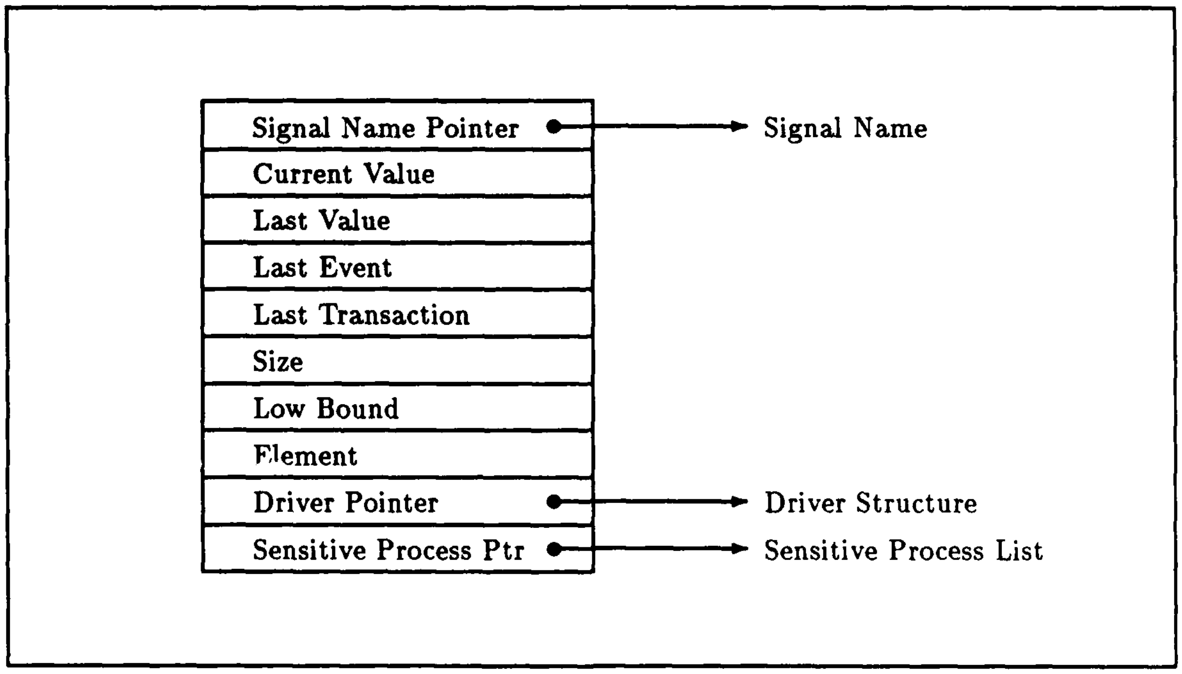

Figure 5-1. Signal Structure

The signal drivers are represented by separate structures that hold information used to determine their respective signal's values. Figure 5-2 depicts the structure used for the driver. All elements of this structure are pointers to other structures. Foremost is the pointer to the signal structure to which the driver is associated. Also included is a transaction pointer that links the projected future values associated with this driver. Lastly, for signals that have multiple drivers, a link to related drivers is present.

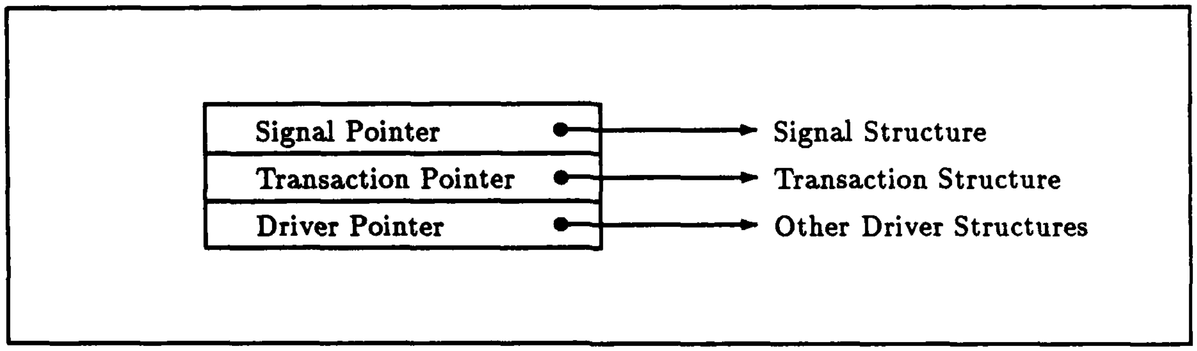

Figure 5-2. Driver Structure

The transaction pointer element for each driver structure points to a list of structures that represent waveform elements. Figure 5-3 shows how the waveform element structure is represented. The waveform elements hold the future values for the driver and the times at which these values are to be assumed. Also included in the waveform element structure is a link to the next waveform element.

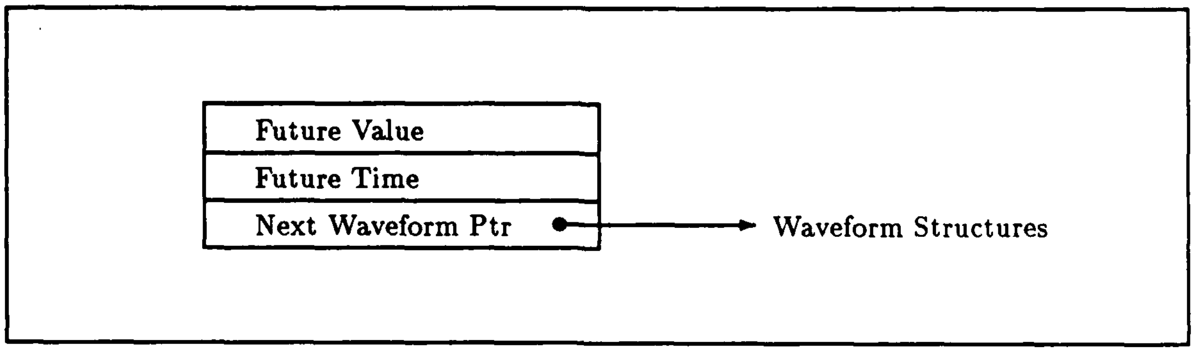

Figure 5-3. Waveform Structure

Certain signal attributes, such as 'Stable, 'Quiet, and 'Delayed are themselves treated as signals by the VHDL model and can be treated as such. They assume characteristics of signals and subsequently have drivers and attributes of their own. Since these attributes are defined by their drivers, they may be used recursively. This means that for signal S, S'Stable'Stable is a valid expression [CLS87b].

5.3.2.2 Variables. Variables are similar to constants in that they represent only a value; they differ from constants in that this value may change. To maintain the current value for a variable, the simulator must create and maintain a data structure for each instance of a variable.

5.3.3 Parse Operation Table. The operation table contains the information about the behavior of the model. This behavior is expressed in terms of VHDL processes with each process utilizing VHDL expressions to describe particular circuit behavior. The simulator must translate these VHDL expressions into C source code that will accurately mimic the VHDL description.

A VHDL process may be a single expression such as a signal assignment statement, or it may be set of sequential statements that form the body of a process statement. In either case, a C function is generated to model the behavior described by the original VHDL source code. As was shown in Chapter III, VHDL supports a range of language constructs to support hardware modeling. The simulator must generate modeling code to model these constructs and their usage.

5.3.3.1 Data Types. Matching instances of VHDL data types is fairly direct. The scalar types integer and floating point are primitive data types for C and can be matched in a one-to-one translation. Enumerated types can be modeled in C using a list of objects. The values declared with a physical type are converted by the analyzer to provide constant values for the simulator.

Instances of a VHDL composite type are split into their component elements since each element represents a unique object. Links are maintained between objects of a composite type to keep order and provide a reference between the objects. For example, a

bit_vector is represented by a linked list of signals. Each signal element has a corresponding driver to provide direct access to the signal. The links are used only when the **bit_vector** is to be operated upon as a whole. An array could have been used to represent VHDL arrays except that VHDL supports user-defined indices for arrays while the indices must begin with zero for all C arrays. As a result, VHDL arrays with negative indices have no equivalent in C.

There are two additional cases where a direct conversion from VHDL is not made. First, type **bit** is modeled using an integer type. Using an integer simplifies the model by being able to use a simpler representation than that for a typical enumerated type. It also makes bit and boolean operations more efficient. Naturally, extra care must be taken to ensure the values other than '0' or '1' are detected. The other instance where a direct conversion is not made is for type **boolean**. Since C does not support flags, type **boolean** is represented by a character in C. A character value of 0 is used for false while all other values are considered true.

5.3.3.2 Primitive Operations. C provides directly equivalent operations to match VHDL's relation, adding, and sign operators (refer to Table 3-1). Other operations such as the boolean operators can be modeled with C functions. Since C supports bit operations, the complete range of VHDL boolean operations can be modeled. Figure 5-4 shows a C function that computes the value for signal C based on the following expression:

C <= A **or** B;

```
char or(s1,s2)
int s1,s2;
{
return ((char) s1 | (char) s2 & 0x01);
}
```

Figure 5-4. C Function to Model **or** Operation

5.3.3.3 Statement Types. The variable assignment and signal assignment are the two statements used to pass values among objects. Since the variable objects only represent a value, the variable assignment statement can be modeled with C's assignment operator. The signal assignment statement, however, entails more than just the assignment of a value to a signal. As was discussed in section 3.3.2, the signal assignment statement projects a waveform that includes both a future value and a future time at which this value is assumed. Adding the delay specified by the delay clause to the current simulation time gives the projected future time. If no delay is specified, delta delay is assumed. In addition, the signal assignment statement specifies whether the inertial or transport delay model is to be followed. The delay model used has a direct impact on determining the projected future waveform for the signal. All of these factors must be considered when modeling a signal assignment statement. For each signal assignment statement, the new waveform element must be created and properly inserted into the driver's waveform. Figure 5-5 shows the C function that would be generated to model the following signal assignment statement:

A <= B **and** C **after** 20ns;

Signals B and C are passed to the function as elements of the array *slist*. The function *Newtrans* is called to create the new waveform element for the driver of signal A. The projected future value is determined by the function *and* as it performs the **and** operation on signals B and C (passed to the function as slist[0] and slist[1]). The projected future value and time are stored in the waveform structure. Lastly, the waveform element is inserted into the waveform by calling the procedure *post_trans*.

VHDL control structures are used to control the execution of procedural statements as would be found in the body of a process statement. C offers a complete set of control structures that map directly to those of VHDL. Table 5-1 lists the VHDL control structures and their C equivalents.

A unique sequential control statement is the **wait** statement. Implementing the **wait** statement is critical in support of the VHDL process model. The **wait** permits the suspension of a process until certain conditions are met. The **wait** statement can be used

```
p14(dlist,slist)
Driver *dlist[ ];
Signal *slist[ ];
{
Transact *Newtrans();
extern TIME simtime;
if(1) {
{
BOOLEAN transport = FALSE;
Transact *newtrans1; /*wave*/
newtrans1 = Newtrans();
newtrans1- >future_time = simtime + 20;
newtrans1- >val = and(slist[0]- >cur_val,
slist[1]- >cur_val);
post_trans(dlist[0], newtrans1,transport);
}
 } else { ; }
```

Figure 5-5. C Source to Model Signal Assignment Statement

in three forms: **wait-on**, **wait-until**, and ***wait-for***. The **wait-on** form suspends a process until an event has occurred on one of a set of specified signals. The **wait-until** suspends a process until a specified boolean expression becomes true. The **wait-for** suspends a process for a specified amount of time.

To implement the wait statement, a method is needed that will suspend a process and continue the same process from the suspended location once the wait condition has

Table 5-1. VHDL/C Control Structures

VHDL	C
if	if
case	case
loop	loop
next	continue
exit	break
return	return
null	null

been met. This can be realized by using a case statement to section the process and direct entry into the process to the proper location. When a **wait** statement is encountered, a variable is set to indicate where the process was suspended. This variable is used by the case statement to continue the process from where it was suspended when the process is re-entered.

After a process has been suspended, it must be scheduled for execution once the wait condition has been met. When a **wait-on** statement is encountered, the suspended process is entered on a special signal sensitivity list. This list references processes by signals to which they are sensitive. If an event occurs on a signal, all processes sensitive to this signal are executed. When a **wait-until** statement is encountered, a dummy signal is created, and it's value is defined by the boolean expression of the wait statement. When an event occurs on this dummy signal, the process is scheduled for execution in the same manner as with the **wait-on** statement. When a **wait-for** statement is encountered, the process is entered on a time-based list of processes to be executed. The time for the process to be suspended is added to the current simulation time to determine the time at which the process is to be continued. When this time is reached, the suspended process is reactivated.

The last of the statement types are declarative statements. VHDL declarative statements are translated into objects by the analyzer and are stored in the symbol table. The analyzer links these objects to an expression in the operation table whenever an initialization clause is present in the declaration. VHDL initialization clauses have a direct C equivalent since C also supports use of an initialization clause in the declaration of variables.

Since VHDL variables are static in the sense that they hold their values during times that a process is inactive, the variable equivalent in the modeling code must also retain its values while the modeling functions are inactive. C supplies the *static* variable which is directly equivalent to the VHDL variable in that it always holds its value during times of inactivity.

5.3.3.4 Operator Definitions. User-defined functions such as those used for overloading and resolution functions are defined in terms of VHDL expressions. The ability to generate modeling code for these functions is dependent upon the ability to generate modeling code for VHDL expressions. The previous sections have shown how C can fully model VHDL operators and statement types. Since these primitive operators and statements are used to compose the various VHDL expressions, there is no problem matching modeling code to the expressions of a user-defined function.

5.3.3.5 Order of Execution. The VHDL model incurs both concurrent and sequential evaluation of expressions. The processes of a model are concurrent and must execute without respect to any order. On the other hand, statements within a process statement must be executed sequentially, in the order in which they are presented. Using a separate C function to model each process allows for both concurrent and sequential execution. Just as each process is to execute without reference to order, each C function is scheduled and executed randomly. The sequential nature of the process statement is modeled by the sequential nature of C expressions within a function.

5.3.3.6 Block Structure. The block structure of the VHDL source code is kept intact as represented by VIA. Knowing the scope of VHDL blocks is important for declaring local variables, establishing design entities, and controlling guard statements. The modeling code is generated in blocks equivalent to that of the original VHDL source code. C uses the { and } brackets to enclose blocks of code and define the scope of execution.

5.3.3.7 Instantiation of Components. Each C function that models a VHDL process receives two parameters. The first is a list of signals to which the process is sensitive, and the second is a list of drivers that are affected by the execution of this process. In this manner, multiple instances of the same component, such as would be found with component instantiation, can be modeled using the same C function.

5.3.3.8 Process Structure. As has been discussed in previous sections, the process is the central element of the VHDL model. It is the basic unit of action from which all descriptions are built. The two statements that represent a VHDL process are the signal assignment statement and the process statement. For each occurrence of these statements in the VHDL description, a C function must be created to model its behavior. Figure 5-5 showed how a C function is used to model a signal assignment statement which represents an AND gate. Other signal assignment statements generate similar functions; the only difference is the driver element which determines the projected future value.

VHDL process statements are a little more involved since they are usually composed of procedural statements that must also be modeled. The body of a process statement may contain signal assignment statements which must be modeled as in Figure 5-5, but it may also have variable and control statements that must also be modeled. These procedural type statements are used to describe behavior in an abstract manner. To correctly model this behavior, the modeling code must directly reflect the VHDL procedural statements. Previous sections have shown that C can directly model VHDL statements; therefore, the generation of the modeling code is a matter of converting the VHDL statements into the equivalent C statements. Figure 5-6 shows a process statement that utilizes some of VHDL's procedural statements. This process statement can be modeled by the C function found in Figure 5-7.

5.4 Simulate

The simulate phase is the portion that is executed to simulate the original design. Code that has been generated to model the circuit design is linked with the simulator kernel and executed. The kernel directs the execution in a manner that follows VHDL's inherent modeling semantics. The steps of this phase include:

1. Create and initialize the model's objects.
2. Invoke all model processes.
3. Update signals.
4. Execute processes sensitive to signals which incurred events.
5. Repeat stems 2 and 3 until simulation is complete.

When simulation begins, all objects of the model are created and initialized. Initial values are determined by the object declaration if an initialization clause was included

```
process (A1)
  variable B1 : integer;
    begin
      if (A1 > 10) then
        A1 <= A1 + 1;
      elsif (A1 > 5) then
        A1 <= A1 + 2;
      else
        A1 <= A1 + 3;
      end if;

      while (B1 < 10) loop
        B1 := B1 + 1;
      end loop;

end process;
```

Figure 5-6. VHDL Process Statement

in the description. Default values are assumed if the initialization clause is not present (see section 3.4.2). The modeling code contains a call to an initialization routine for each object of the model. This routine dynamically allocates the structures needed to create the object and initializes the representation to the proper initial value.

Once the objects are declared and initialized, all processes of the model are executed. Process execution is accomplished by calling all the C functions that were generated to mimic the individual processes of the model. During execution, the processes will cause transactions to occur for each driver internal to its process. These transactions generate a projected future value for each driver and in turn determine the value of their respective signals. A signal may have more than one driver, such as may be the case for a *wired-or*, and the projected future value must be resolved using the signal's associated resolution function (see section 3.2.5.2). The AVE Analyzer/Simulator only supports single drivers for each signal.

When a projected future value is generated, its waveform element is inserted into the list of projected future values for that driver (see section 3.4.3.2). This list is ordered by simulation time, with the earliest projected waveform elements coming first. The projected

```
p54(dlist,slist)
Driver *dlist [ ];
Signal *slist [ ];
{
  Transact *Newtrans();
  extern TIME simtime;
  static int v47 = 0; /* B1 */
  if(GT(slist[0]- >cur_val, 10))
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1- >future_time = simtime + 0;
    newtrans1- >val = ADD(slist[1]- >cur_val, 1);
    post_trans(dlist[0], newtrans1,transport);
  }
  else if(GT(slist[2]- >cur_val, 5))
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1- >future_time = simtime + 0;
    newtrans1- >val = ADD(slist[3]- >cur_val, 2);
    post_trans(dlist[1], newtrans1,transport);
  }
  else if(1)
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1- >future_time = simtime + 0;
    newtrans1- >val = ADD(slist[4]- >cur_val, 3);
    post_trans(dlist[2], newtrans1,transport);
  }

  while(LT(v47, 10))
  {

    v47 = ADD(v47, 1);
  }


}
```

Figure 5-7. C Function to Model VHDL Process Statement

future values are stored in a waveform element structure that holds the value along with the projected future time and a link to the next waveform element. Figure 5-8 depicts this signal/waveform relationship.

The time of the first projected future value for a driver determines its position in the ***time queue.*** The time queue is a global list of times when one or more transactions are scheduled to take place. Each entry in the time queue may have one or more drivers whose first transaction is scheduled to take place at this entry's simulation time. This list of drivers is called an ***event list*** and is depicted by Figure 5-9.

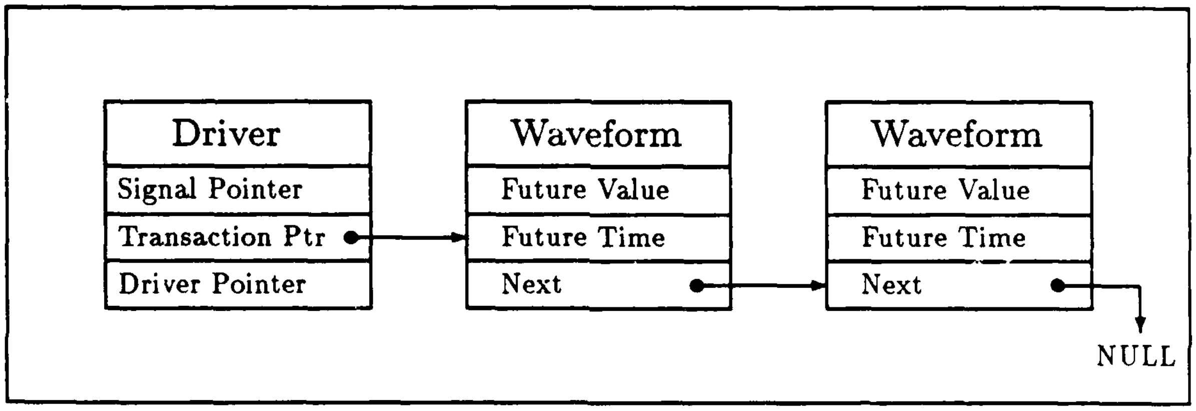

Figure 5-8. Driver with Projected Waveform Elements

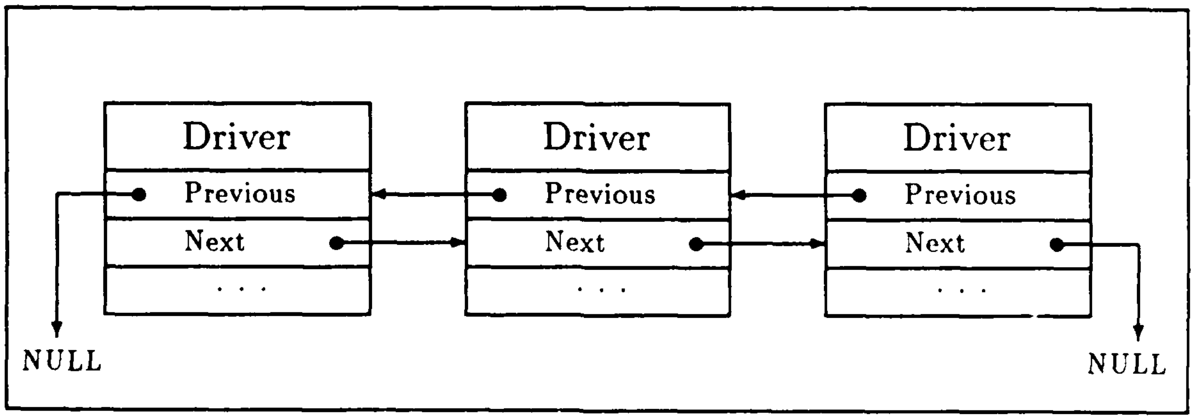

Figure 5-9. Event List

The time queue is a binary tree structure with each node pointing to an event list. The order of the time queue is based upon the transaction time of each event list. Figure 5-10 depicts a time queue with an associated event list having a projected future simulation time of 50ns. When the simulation time reaches 50ns, the value of each driver is assumed by that driver's associated signal. If a signal assumes a value that is different from its original value, then an event has occurred. The occurrence of an event triggers the activation of any processes that are sensitive to the signal which has incurred the event. This forces an evaluation of all signals that are dependent upon the signal that had the event. This is the cause-effect relationship among objects in the VHDL model.

5.5 Summary

This chapter has presented the implementation details about the AVE simulator. The Build package was discussed from the point-of-view of how C modeling code is generated to model the behavior described by the VHDL source code. The Simulate package was discussed in regards to the execution of the simulation model. The next chapter will compare the results of the implementation against the original requirements and design objectives.

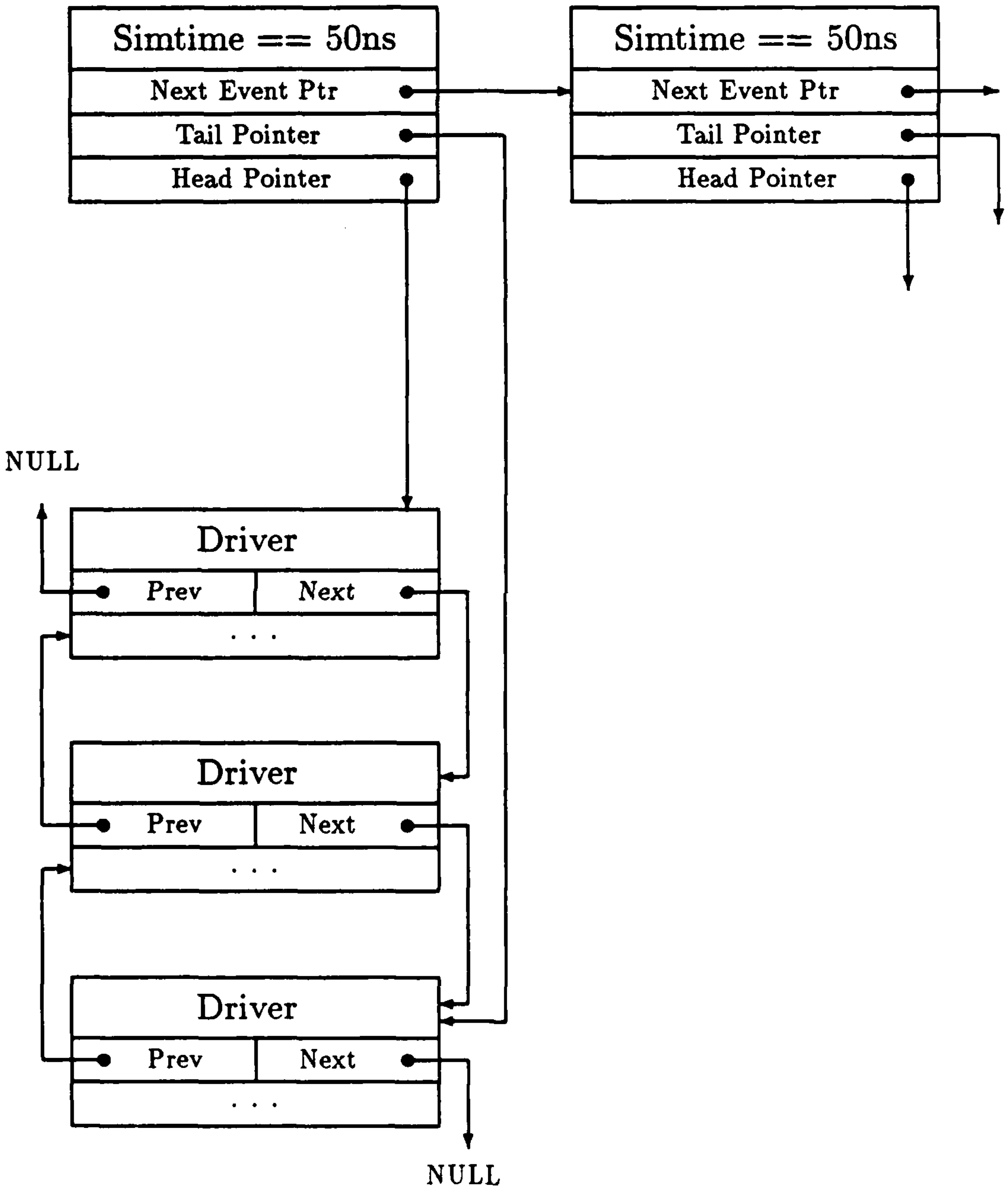

Figure 5-10. Time Queue with Event List

VI. Analysis and Results

6.1 Introduction

This chapter reviews the accomplishments of this thesis. The design and implementation of the simulator is compared against the original requirements and design objectives. Next, the methods used to test the software are discussed in regards to validating the requirements and verifying proper software operation. Finally, performance measures using the simulator on different hardware platforms are presented.

6.2 Requirements

The requirements for the simulator are listed in Chapter IV. These requirements were divided into five areas: general, functional, implementation, operational, and system.

6.2.1 General Requirements. The general requirements dealt with miscellaneous considerations about the simulator that were not directly software related. These included documentation such as a User's Manual, an Installation Guide, and a Programmer's Guide. These three documents were completed to provide information on how to operate, maintain, and install the simulator. Copies of these manuals can be found in Appendices A-C.

6.2.2 Functional Requirements. The functional requirements relate to the proper operation of the simulator in regards to accurate simulation of VHDL. Included in these requirements were proper modeling of the VHDL language features along with the inherent simulation characteristics of the VHDL model.

The final version of the simulator fulfills the stated functional requirements. Validation to show that the software meets the functional requirements is discussed in more detail in section 6.4.

6.2.3 Implementation Requirements. The implementation requirements focused on how the simulator was to be constructed. These were directives to be followed to provide standards and to form a basis from which design decisions can be made. The first requirement was that the simulator be precompiled and event-driven. Research showed that a precompiled, event-driven simulator was most efficient for use in simulating large digital designs. In addition, the VHDL model's cause-effect relationship of signal propagation makes event-driven simulation the logical choice for implementation.

The next implementation requirement dealt with how the software was to be implemented to support a precompiled simulator. The precompiled notion implies a simulator kernel to direct and control the simulation process. Modeling code for VHDL must be generated and linked to the kernel. This process requires two steps—one to generate the modeling code and one to execute this code via the simulator kernel. Following this methodology meant implementing the software to model these two process steps. The Build and Simulate packages were the outgrowth of this design decision.

The next requirement concerned the platform from which the simulator was to operate. As was discussed earlier, the AVE simulator was targeted for use in an educational environment. Since the UNIX operating system is common to many of these environments, it was chosen as being the primary operating system for the simulator development. The requirements that the simulator operate on a VAX 11/785, an ELXSI 6400, and a Sun workstation were made because these systems are commonly found at universities. Operation of the simulator on each of these computers was successful. The simulator was tested while running under UNIX bsd versions 4.2 and 4.3 with no noted problems. Meeting the requirement of operating under MS-DOS on a PC compatible computer was met, but with reduced capabilities for the simulator. The limitations with operating under MS-DOS were due to the 64 kilobyte limitation for data structures. This data structure limitation means that the simulation models cannot be built for VHSIC class chips; however, simulation of common components such as a full adder can be efficiently executed.

The requirement to enhance portability and expandability was met by providing a design of highly cohesive modules with the isolation of machine/operating system dependent code. Changes necessary to port the software to other computer environments should be minimal. The Programmer's Guide provides a more detailed look at how this can be accomplished.

6.2.4 Operational Requirements. The operational requirements were those that affected the operation of the simulator from a user's standpoint. These included operational standards and options to be provided. The simulator features that directly relate to the operational requirements are described below. A more comprehensive discussion of the simulator operation is presented in the User's Manual (see Appendix A).

The simulator operates in one of two modes: batch and interactive. In either mode, the user can specify the start time, the termination time, and the output file into which the simulation results are to be stored. In batch mode, the simulator runs to completion based upon the specified timing parameters. In addition, the user may specify a vector file as input to the simulation. This file allows the user to specify signal values to be used during the simulation. These values can be used to initialize the circuit, provide input stimulus to the circuit, or to analyze the circuit under "what if..." conditions. The user may also specify a trace file that is used to specify which signals are to have their values recorded during the simulation. Trace options include recording signal values with each transaction or event. Specifying the signals of interest and how they are to have their values recorded will help reduce some of the overhead of tracking signal values and improve simulator efficiency.

6.2.4.1 Interactive Features. In the interactive mode, the user has the option of setting breakpoints, viewing signal values, changing signal values, setting and locking a signal to a value, and unlocking a previously locked signal. These options are particularly useful to engineers who are using VHDL to test and debug their designs. The user can control execution with commands to run the simulation to the next breakpoint, run until the next time in the time queue, run until completion, or to terminate the simulation without further processing.

6.2.5 System Requirements. The system requirements cover the requirements to be met to interface the simulator with its environment. This includes interfaces between the simulator and AVE tools and the interfaces between the simulator and the host machine. The VHDL Intermediate Access is used as the interface medium for tools within AVE. It holds the syntactic and semantic information of the VHDL description in an abstract syntax tree structure to facilitate parsing by other tools. The requirement to use C library functions was made to reduce the amount of original code needed for the simulator. Most C programming environments include a set of common C functions; therefore, their use should not affect portability.

6.3 Design

The design specifications were based upon those provided by Lt Kodama in his work [Koda87]. Changes in the design pertained to extensions and modifications to the existing simulator that were necessary to implement new features and to make the simulator conform with IEEE Standard 1076. Changes to model objects were made to extend existing features such as implementing signal attributes. New model objects were created to add features such as composite types. The design of the simulator kernel had to be modified slightly to accommodate the modeling guidelines presented by IEEE Standard 1076.

6.4 Testing

Testing for the AVE simulator was accomplished for two reasons: to validate the requirements and to verify the correctness of code execution. Validation testing was conducted to ensure that the software functions in accordance with the specified requirements. This type of testing is mainly black-box or functional type testing that ensures that the output of the system is correct for the type of input used. The main requirement for the simulator is to correctly model circuits described using VHDL. The simulator must be able to correctly model each feature of VHDL as is defined by IEEE Standard 1076. Additional requirements such as full compatibility with UNIX, efficient use of CPU time, and efficient use of disk space were also validated. The criteria for efficient performance is be based upon performance of similar systems such as Intermetric's VHDL environment.

Verification testing was used to monitor code execution so that errors in the design logic and code implementation could be identified. This testing was conducted at the software module level to test the structure and code execution within the module for correct operation. Test cases were created to detect errors with the module interfaces, local data structures, code execution and control flow, boundary conditions for data, and error handling. Verification testing was also conducted at a higher level to test the integration of the software components. Critical to this integration is the interface specification for each module. Although the interface requirements for each module was tested during unit level testing, it was tested again during integration testing on a much larger scale.

The existing simulator software was complete in the sense that it provided complete functionality for the simulation of a subset of VHDL. Extending the simulator required the creation of new routines along with the modification of existing routines. With an existing prototype simulator from which to work, the new software could build on existing software and be developed in a rapid prototype approach. This approach follows the strategy outlined by the spiral software development cycle [Boeh88]. In this manner, features were designed, implemented, and tested individually. When one feature was fully implemented and tested, work on the next feature was begun. New features added to the simulator were implemented into the existing software framework and fully tested under actual operational conditions.

At the completion of each spiral of the development cycle, a series of tests were conducted for validation and verification. Implementing language features incrementally permitted the isolation and testing of each feature as it was implemented. In addition, a set of regression tests were conducted at the completion of each spiral cycle to ensure that new feature implementations had no adverse effect on other simulator operations.

At the conclusion of the software development, a suite of tests were conducted to verify proper integration of all language features. Although code execution was monitored for anomalies, the primary purpose for these tests was to ensure that the simulator operated in accordance with IEEE Standard 1076 for VHDL. The majority of test cases used during this test came from the VHDL repository at SIMTEL20. This repository contains a comprehensive suite of test cases that exercise each feature of VHDL. The test cases that contain features not supported by the AVE simulator were not run.

6.5 Performance

The relative efficiency of the simulator's operation in regards to resource usage was of great concern during development. Targeting the AVE environment for the educational community meant that its toolset must operate in environments that may have very limited resources. The concerns centered on the amount of disk space and time needed to build and execute the simulation model.

To measure the simulator's resource usage with varying host computers, a simulation model for a full adder was built and executed on a VAX 11/785, an ELXSI 6400, and a Sun workstation. Table 6-1 shows the disk usage and operational times for these tests.

Table 6-1. AVE Simulator Resource Usage

Phase	ELXSI		VAX 11/785		Sun	
	CPU Time (Sec)	File Size (Bytes)	CPU Time (Sec)	File Size (Bytes)	CPU Time (Secs)	File Size (Bytes)
Build	0.1	9397	1.2	9397	2.7	9397
Compile	3.0	43302	3.3	37888	3.9	38943
Simulate	0.1		0.3		0.4	

6.6 Summary

This chapter reviewed the outcome of the development of the AVE simulator. This review compared the final product against the original requirements for the software. It also presented a look at developmental aspects such as the design, testing, and performance measures and their impact on the project.

VII. Conclusions and Recommendations

7.1 Introduction

The purpose of this thesis was to analyze VHDL in regards to its suitability for hardware modeling and to use this knowledge in developing a simulator to complement the CAD tools within the AFIT VHDL Environment. This chapter presents conclusions reached during this effort and suggest future research areas for simulation with VHDL.

7.2 Conclusions

The groundwork for this thesis was the study of VHDL and its support for hardware simulation. As discussed in Chapter III, VHDL provides both the explicit and implied constructs to model hardware structure and behavior. In addition, VHDL provides the needed flexibility through user-defined constructs and operations to describe unique structure and behavior. VHDL is not limited to any hardware technology or any design style. It supports a range of hardware descriptions, from structural to behavioral, through varying levels of abstraction. Its acceptance as a standard hardware description language will promote the sharing of information among design engineers and further the advances of integrated circuit technology.

The implementation of the AVE simulator was successful in that it meets the established requirements. Developing a simulator that adheres to the new IEEE Standard 1076 is a big step in keeping the AVE abreast with current technology. The simulator's portability to other operating environments will facilitate the distribution of AVE tools to the education community and promote the use of VHDL.

7.3 Recommendations

7.3.1 Parallel Simulation As integrated circuit technology continues to progress and chip integration become more dense, simulation of hardware becomes more time consuming on conventional, single processor computers. One method to overcome the limitations of a single processor computer is to use parallel simulation. The VHDL process model maps well to a parallel implementation. Since each VHDL process executes independent of one another, they can be assigned to separate processors for execution. Spreading the workload across a number of processors would greatly speed simulation.

7.3.2 AFIT VHDL Environment. For the AFIT VHDL Environment to continue to grow, additional CAD tools must be researched and implemented. Tools that would have a direct impact on the AVE simulator include a design library management tool and a graphical interface. In addition, developing environment tools such as a syntax-directed editor would improve efficiency for creating VHDL descriptions.

7.3.3 Library Management Tools. For the AFIT VHDL Environment to continue to grow in an integrated fashion, a library manager to organize and control design entities is needed. Currently, the AVE analyzer produces VIA from VHDL source code to be used by other AVE tools; however, the use and control of the design entities is left up to the individual tools. A library manager would control the use of existing design entities and promote component reuse. The existence of a library manager would also improve such capabilities as separate compilation of design entity components, and component instantiation.

7.3.4 Graphics Back-end A possible extension to the current simulator is a graphical interface to represent the simulation results. The AVE simulator provides simulation reports in a text-only format. To be able to visually see the circuit under simulation would increase the understanding of the circuit's operation.

7.3.5 Syntax-Directed Editor Development of a syntax-directed editor for VHDL would serve two purposes. First it would provide a tool that would improve the efficiency of developing VHDL descriptions by helping eliminating time lost to correcting syntax errors. Secondly, a syntax-directed editor would make learning VHDL easier. With a need to promote the use of VHDL, any tool that may encourage its use and widen its acceptance would be very valuable.

7.4 Summary

The importance of having a standard HDL such as VHDL is to promote compatibility across a wide spectrum of digital circuit designs and to help further the advancement of integrated circuit technology. To achieve these goals, use of the language must be promoted by developing and distributing VHDL tools for the engineering and educational communities. This type of promotion was the motivation for developing the AVE simulator. Its distribution to universities and other interested parties will introduce VHDL to researchers and students and promote a better awareness of the language.

Appendix A. User's Manual

A.1 Introduction

The AFIT VHDL Environment (AVE) is a set of CAD tools that support the use of the VHSIC Hardware Description Language (VHDL). Central to this environment is the AVE analyzer and simulator. The analyzer is used to parse VHDL descriptions and check for syntax and static semantic errors. If no errors are found in the VHDL description, the analyzer generates an intermediate form that represents the model described by the original VHDL source code. This intermediate form, called VHDL Intermediate Access (VIA), is a series of records that contain information about the original design.

The VIA is used as input to the AVE simulator. The simulator parses the information contained in the VIA and generates a model for simulation. The simulator is executed in two phases: Build and Simulate. The Build phase generates C source code that is used to model the behavior of the original VHDL description. The Build phase is complete when the modeling code has been compiled and linked with the simulator kernel for execution. The Simulate phase is begun with the execution of the model created during the Build phase.

Use of the AVE analyzer and simulator as described in this manual are for use in a UNIX environment. The details of how to install the software are explained in the AVE simulator's Installation Guide.

A.2 Getting Started

To use the AVE analyzer and simulator, you must have access to their respective files. In a UNIX environment, the AVE tools will reside in a single directory that is accessible to all. Executing the programs from this directory will require that a path be set to this directory by using the UNIX *set path* command. For example, on AFIT's bsd system, the user gains access to the tools by entering the following at the UNIX prompt:

```
set path = ($path /cad/vhdl)
```

This command gives access to the AVE directory, but only for the current login session. If the AVE tools are to be accessed on a regular basis, the *set path* command can be added to the user's *.login* file (See the UNIX manual for additional information).

A.3 Analyze

To analyze a VHDL description, the user issues the following command:

vhdl *filename* [-l | -n]

where *filename* is the name of the VHDL source file. VHDL source files to be used in the AVE environment must have a **.vhd** extension, but the user may specify the filename with or without the extension on the command line.

The two options available on the command line pertain to list file generation. The list file will identify errors detected by the analyzer. The list file options are:

–l	Produce a list file whose name has the same root name as the source file but with a **.lst** extension. This file will be placed into the current working directory.
–n	Produce no listing.

If no option is specified, then the analyzer defaults to displaying the listing to the user's terminal.

In addition to the list file, the analyzer generates the VIA file to be used by the simulator. The filename for the VIA file will have the same root name as the source file, but with a **.via** extension. This file will be placed in the user's current working directory.

A.4 Simulate

There are two steps to simulating a VHDL description: Build and Simulate. But before these steps can be taken, the VHDL description to be simulated must be analyzed first. The analyzer produces a VIA file that is used as input to the simulator.

A.4.1 Build. To build a simulation model, the user must enter the following command:

build *filename*

where *filename* is the name of the VIA file. All VIA filenames must have a **.via** extension, but the user may specify the file with or without this extension on the command line.

The user will be notified when the build is complete. The Build produces a file that is used to simulate the original VHDL description and places it in the user's current working directory. This file will have the same root name as that of the VIA file, but with a **.sim** extension.

A.4.2 Simulate. To execute the simulator for a given model, the user's enters:

filename [−b|−i][−o ofile][−s stime][−t ttime] [−v vfile][−n nfile]

Where:

filename	is the name of the simulation model
−b	specifies batch mode
−i	specifies interactive mode
−o	identifies the output file used for the simulation report
−s	identifies the simulation start time
−t	identifies the simulation termination time
−v	identifies the vector input file
−n	identifies the no-trace file

Default values for these options are:

Simulation mode	= "batch"
Output file	= "output.trn"
Start time	= 0 ns
Termination time	= 1000 ns

A.4.2.1 Batch Mode. If no options are present on the command line, the simulator will display a help screen of options. If a mode option is not selected with other options, the simulator will default to batch mode. The batch mode will execute the simulation unattended under the specified and/or default options.

A.4.2.2 Interactive Mode. When the interactive mode is chosen, the user can direct control of the simulation through the use of a menu-driven *interface. Figure* A-1 shows the menu with the interactive options available.

```
? –  Help
b –  Set a breakpoint
v –  View signal Values(s)
c –  Change signal value(s)
l –  Set and lock a signal to a value
u –  Unlock a previously lock signal
r –  Run the simulation to next breakpoint or completion
n –  Run the simulation until time advances
x –  Terminate the simulation
```

Figure A-1. Interactive Simulation Menu

Help. This option displays information about the options.

Set a Breakpoint. The user may specify a time at which the simulation will suspend. Suspending the simulation permits the user to use other interactive options such as viewing and changing a signal value.

View Signal Value(s). Anytime the simulation is suspended, the user may view the value of any signal in the model. The user is prompted for the name of the signal to be viewed. This name must be the same as was used in the original VHDL description.

Change Signal Value(s). When the simulation is suspended, the user may change the value of any signal in the model. The user is prompted for the name of the signal to change and is then prompted for the new value.

Set and Lock a Signal to a Value. In addition to changing the value of a signal, the user may lock the *signal to this value*. This means that the signal will always assume this locked value and will never change. This provides a means of doing fault testing.

Unlock a Previously Locked Signal. The unlock option simply reverses the effects of the lock option. When the signal is unlocked, it regains normal signal characteristics and assume new values when applicable.

Run the Simulation to Next Breakpoint or Completion. This option permits the user, to execute the simulation from one breakpoint to the next.

Run the Simulation Until Time Advances. This option is used to execute the simulation until the next transaction occurs. At this time, the simulation is suspended, and the interactive menu is redisplayed.

Terminate the Simulation. The user selects this option to abort the simulation at its present suspended state.

A.4.2.3 Start and Stop Times. The simulation time will begin at the specified start time and will stop at the specified termination time. If these values are not specified on the command line, the simulation will assume 0ns at the start and will terminate at simulation time 1000ns.

A.4.2.4 Vector Input Files. A vector input file is used to specify values for signals of a model. These values can be used to initialize signals to a certain value or to change signal values during simulation. The vector input file provides a simple method to generate the inputs to a component as might occur during normal operation. The structure of the vector input file is shown in Figure A-2.

```
 2  A  B
20  0  1
40  1  0
60  1  1
80  0  1
-1
```

Figure A-2. Vector Input File

The first line of the example specifies the number of signals whose values are to be specified, along with the names of these signals. The signal names must be the same as those used in the original VHDL description. The following lines specify a simulation time and values to be assumed by the signals at this time. The order in which the signal values are presented must match the order of the signal names on the first line. The last line of the example shows a −1. This value signifies an end-of-file.

A.4.2.5 Trace File. The trace file is used to specify which signals are to be traced and reported in the transaction output file. Without a trace file, the simulator defaults to all signals being traced, with output being generated for every signal transaction. The user may narrow the analysis by specifying the signals to be traced. Figure A-3 shows an example trace file. The trace options are used to specify report generation for each signal transaction or for each signal event. The keyword ***event** must precede a list of signals that are to be traced by event. The keyword ***trans** must precede a list of signals that are to be traced by transaction. As the example shows, separate ***event** and ***trans** lists may appear within the same file.

```
*event
A
B C
*trans
D E
*event
F
G
```

Figure A-3. Signal Trace File

Appendix B. Installation Guide

B.1 Introduction

This guide gives instructions on how to install the AVE simulator software. These instructions assume a UNIX environment where the simulator is located in a unique directory that may be accessed by all.

B.2 Directory Structure

A unique directory should be created to establish the AVE. This directory be accessible by all who plan to use the AVE tools. The immediate AVE directory will hold the executable files that need to be accessed by the users. All support files will be located in directories below this main directory. Figure B-1 shows the structure of the AVE directory. A UNIX shell called *installave* can be used to build the AVE directory. This file should be found with the other simulator files.

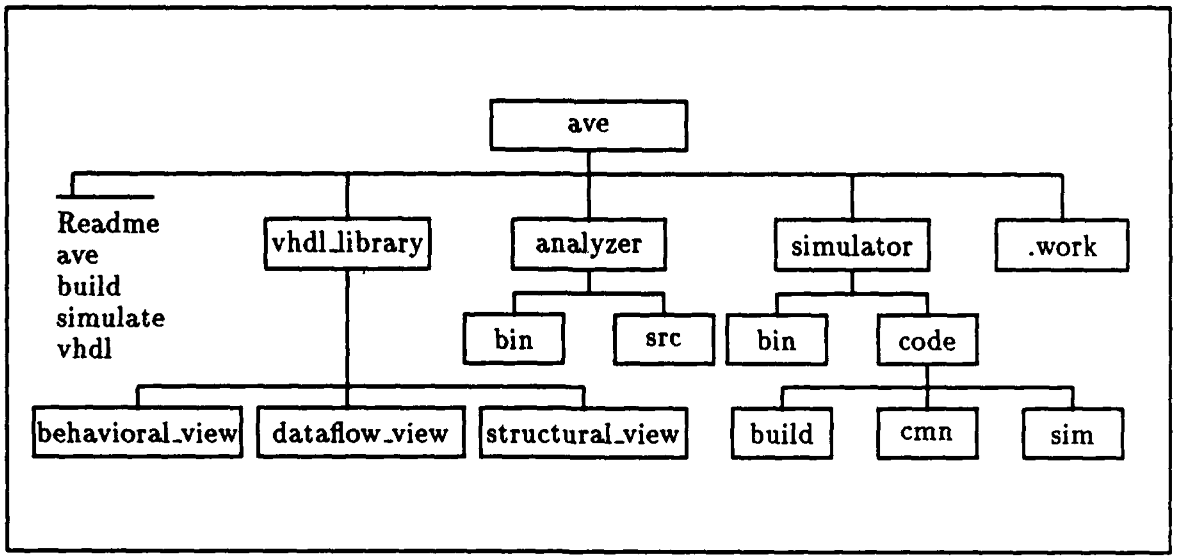

Figure B-1. AVE Directory Structure

The AVE directory should contain the following files:

Readme	Explains the files/subdirectories of the current directory.
ave	This is a UNIX script that provides a menu driven interface to the AVE tools and utilities.
build	This is a UNIX script that invokes the build phase of the simulator
simulator	This is a UNIX script that executes the AVE simulator.
vhdl	This is a UNIX script that invokes the AVE analyzer.

The AVE directory should contain the following subdirectories:

.work	This directory is used to store miscellaneous work files used by the analyzer or simulator.
analyzer	This directory contains all the analyzer files.
simulator	This directory contains all the simulator files
vhdl_library	This directory holds a library of VHDL descriptions.

The analyzer directory should contain the following subdirectories:

bin	Directory for the analyzer's executable files.
src	Directory for the analyzer's source files.

The simulator directory should contain the following subdirectories:

bin	Directory for the simulator's executable files.
code	Directory for the simulator's source and object files.

The code directory under simulator should contain the following subdirectories:

build	Directory for the simulator's build routines.
cmn	Directory for the simulato.'s common routines.
sim	Directory for the simulator's simulate routines

B.3 Copying Source Files

Once the AVE directory structure has been established, the source code and supporting files for the simulator can be moved into the AVE directories. These files should come from an archived set of baselined files.

The following files should be placed into the /simulator/code/build subdirectory of the AVE directory:

Readme	bldsim04.c	build.c	makefile
blddummy.c	bsimmhb.c	cleanup.c	read_via.c
bldsim01.c	bopeval.c	cmdlineb.c	vhdl.h
bldsim02.c	bopwalk.c	f_insig.c	vhdlyacc.h

The following files should be placed into the /simulator/code/cmn subdirectory of the AVE directory:

Readme	new_stru.c	site_unique.h	strfcn.c
makefile	sim_stru.h		

The following files should be placed into the /simulator/code/sim subdirectory of the AVE directory:

Readme	findevti.c	read_tra.c	simmain.c
addevit.c	flib.c	read_vct.c	simtime.c
buildsim	intmenu.c	report.c	simulate.c
cmdlines.c	intopt.c	rmevliit.c	up_timeq.c
del_tran.c	makefile	sfindsig.c	updateeq.c
dohelp.c	posttrns.c	sim100.c	upprojwv.c
dque.c	procme.c		

B.4 Initialization

After all the simulator files have been moved into their proper directories, some initialization processing must be accomplished. This process has been automated and can be accomplished by executing the UNIX script called *installsim*. This file should be found with the other simulator files. A copy of this script should be placed into each of the three simulator code subdirectories:

```
<ave>/simulator/code/build
<ave>/simulator/code/cmn
<ave>/simulator/code/sim
```

where `<ave>` was the directory chosen for AVE.

The *installsim* script will compile the source code and produce executable files. It will also handle other initialization processes that will customize the software to work from the new AVE directory.

The first step towards initialization is to enter the `<ave>/simulator/code/cmn` subdirectory and execute the *installsim* script.

The second step is to enter the `<ave>/simulator/code/build` subdirectory and execute the *installsim* script.

The last step is to enter the `<ave>/simulator/code/sim` subdirectory and execute the *installsim* script. Installization is complete. The AVE simulator is now ready to use.

Appendix C. Programmer's Guide

C.1 Introduction

This guide explains the concepts and operation of the AVE simulator from a programmer's perspective. This information should be useful for maintaining and enhancing the existing software. This guide presents an overview of the system and then looks at each portion in more detail.

C.2 System Overview

The AVE simulator is composed of two packages: Build and Simulate. The Build package generates C source code to model the design, compiles the modeling code, and links it to the simulator kernel. The Simulate package is the combination of the compiled modeling code and the simulator kernel. The kernel contains routines that control the simulator model and provide access to the user.

Another portion of the simulator software is the group of *common* files. These include *header* files and common functions used by both phases of the simulation. These files are discussed first.

C.3 Common Files

C.3.1 sim_stru.h This file contains all the common structures used by both the build and simulate packages. Included are the signal, variable, driver, transaction, event queue, time queue, time list, process table, and VIA structures. Also included are definitions for constants and aliases used by all routines.

C.3.2 site_unique.h This file contains definitions for constants and aliases that may vary from installation to installation. These items were separated from *sim_stru.h* to make changes easier when porting the software.

C.3.3 new_stru.c This file contains routines to dynamically create the structures that were defined in *sim_stru.h*. These routines are called during simulation as each structure is needed. These routines also initialize the structures using the values passed by the simulator kernel.

C.3.4 strfcn.c This file contains common string functions used by both packages. Although these routines may appear in a C library at some installations, providing them as part of the simulator software will avert problems that would arise when the software is ported to an environment that does not provide these functions as library routines.

C.4 Build Package

The Build package reads VIA and creates the data structures and modeling code for the simulation. The first step of the Build phase is to read the VIA into internal data structures for parsing. The VIA contains information in the form of a symbol table, an operation table, and a string table. The symbol table contains information about objects, such as signals and variables, used in the circuit description while the operation table contains information about the circuit operation. The symbol table also contains string literals used in the circuit description.

Using the information contained in the VIA, the Build phase generates C source code that, when compiled and linked with the simulator kernel, will mimic the behavior of the original VHDL description. Six files are generated by the build phase: sim01.c, sim02.c, sim03.c, sim04.c, sim05.c, and simmhb.c . These are the files that contain the modeling code.

Following is a list of files that form the Build package. A description accompanies each file listing the routines contained in the file along with each routine's function.

C.4.1 blddummy.c This file contains one routine: build_dummy_signal. Its purpose it to build dummy signals for elements of the model such as *guard* statements that assume signal characteristics, but are not actually signals. A dummy signal is also built for the expression of a **wait-until** statement. This makes it possible to monitor the expression and reactivate the suspended process when the expression becomes true. The dummy signals are put into the sim05.c portion of the modeling code.

C.4.2 bldsim01.c This file contains two routines: bsim01a and bsim01b. Their function is to generate the sim01.c portion of the modeling code. sim01.c contains the routine sim_initialize which is used during execution of the simulator to initialize the model. bsim01a generates some of the declaration for sim_initialize. The declarations include the signal array and process table array. bsim01a also generates statements to declare the pointers to each signal structure. bsim01b generates statements to declare pointers to each driver of the model. bsim01b also generates the statements that dynamically allocate the signal and driver structures.

C.4.3 bldsim02.c This file contains two routines: bsim02a and bsim02b. Their function is to generate the sim02.c portion of the modeling code. sim02.c contains declarations for the sen_proc array for each signal. This array points to all processes that are sensitive to a given signal. sim02.c also contains a table of the processes in the model.

C.4.4 bldsim04.c This file contains two routines: bsim04a and bsim04b. Their function is to generate the sim04.c portion of the modeling code. sim04.c contains *extrns* and *includes* to be used by the modeling code. sim04.c also contains the Signal and Driver declarations to be used by the modeling code.

C.4.5 bopeval.c This file contains one routine: op_table_eval_node. Its function is to parse the operation table portion of the VIA and generate modeling code. As the modeling code is generated, it is placed into the file, sim03.c .

C.4.6 bopwalk.c This file contains one routine: op_table_walk. Its function is to direct the VIA operation table parsing. VIA stores operation information in a binary tree structure with common information occupying a branch. The tree structure of VIA permits it to be parsed in a recursive fashion. op_table_walk directs the recursive parsing down the proper branches.

C.4.7 bsimmhb.c This file contains one routine: bsimmhb. Its function is to generate the simmhb.c portion of the modeling code. simmhb.c contains declarations for the modeling code that are unique to the particular model to be simulated.

C.4.8 build.c This file contains one routine: main. This routine is the driver for the build phase. It calls the subordinate routines to build the modeling code and controls the opening and closing of the files that hold the modeling code.

C.4.9 cleanup.c This file contains one routine: cleanup. Its function is to perform cleanup activities whenever the program is to be exited.

C.4.10 cmdlineb.c This file contains one routine: cmdlineb. Its function is to parse and validate the parameters passed from the command line. This routine sets a file pointer to the VIA file specified by the user on the command line.

C.4.11 f_insig.c This file contains four routines: ins_sig_ref, ins_sigalf_ref, build_sig_lists, and get_object_attributes. The first three routines are used to parse the symbol table portion of VIA and build a linked list of signals. The get_object_attributes is used to extract attribute information from the symbol table when a signal is found.

C.4.12 read_via.c This file contains one routine: read_via. Its function is to read the VIA records stored on disk into an internal representation.

C.5 Simulate Package

The simulate package contains the simulator kernel. These routines control the simulation cycle, record circuit transactions and events, and provide control of the simulation to the user.

C.5.1 addevit.c This file contains one routine: add_event. This routine adds an event to the event list. This is accomplished by inserting the driver of the signal that incurred the event into the event list.

C.5.2 cmdlines.c This file contains one routine: cmdline. This routine parses and validates the parameters passed from the command line. Values such as start time, stop time, vector input filename, trace filename and simulation mode are saved and passed to other executive routines of the simulator kernel.

C.5.3 del_trans.c This file contains one routine: del_trans. This routine is called to delete a transaction from a driver's transaction list. This routine is called when the transaction has taken place, or when another transaction preempts a previous transaction.

C.5.4 dohelp.c This file contains one routine: dohelp. This routine provides additional help information about the simulator's interactive help commands.

C.5.5 dque.c This file contains two routines: dque and d_sigval. These two routines provide debugging information during simulation. They generate information about the queues and signals and store this information into a file called *output.out.*

C.5.6 findevti.c This file contains one routine: find_eventq_time. When a transaction occurs, the event queue must be searched for an entry that matches the transactions projected future time. find_eventq_time searches the event queue for a given time, and if one is not found, a new entry into the event queue is made for the new time.

C.5.7 flib.c This file contains the C functions used to model VHDL operations. Included are functions to model VHDL logical, relational, and arithmetic operations.

C.5.8 intmenu.c This file contains one routine: intmenu. This routine is the driver for the interactive menu. It provides the menu and invokes other interactive routines based upon user input.

C.5.9 intopt.c This file contains seven routines: set_bp, change_sig, view_sig, dispmenu, showtime, lock_sig, and unlock_sig. These routines are used to provide the interactive features.

C.5.10 main.c This file contains one routine: main. This is the driver routine for the simulator kernel. This routine initializes the model and invokes the simulation process.

C.5.11 posttrns.c This file contains two routines: post_trans and post_new_trans. These routines are used together to add a new transaction to the proper driver waveform.

C.5.12 procme.c This file contains two routines: mark_proc_tbl and exec_proc_tbl. When an event occurs for a signal, all processes sensitive to that signal must be marked for execution. mark_proc_tbl accomplishes this task as it searches a signal's sensitive process list and adds these processes to a list to be executed. exec_proc_tbl is called to invoke all processes marked for execution.

C.5.13 read_tra.c This file contains one routine: read_trace_file. This routine is called whenever a trace file is specified on the command line. It's function is to read the trace file and flag the signals that are to be traced.

C.5.14 read_.vct.c This file contains two routines: read_vector_file and post_change. This routine is called whenever a vector file is specified on the command line. It's function is to read the vector file, and assign the input values to signals within the model.

C.5.15 report.c This file contains one routine: report_trans. This routine is called to generate the output report on the transactions as they occur during simulation.

C.5.16 rmevliit.c This file contains one routine: rm_event_list_item. This routine is called to remove drivers from the event list.

C.5.17 sfindsig.c This file contains one routine: find_signal. This routine is used to search a list of signals based upon a given signal name. When the signal is found, a pointer to that signal is returned.

C.5.18 sim100.c This file is used to merge the modeling code files. Each modeling code file is *included* into sim100.c for compilation.

C.5.19 simtime.c This file contains two routines: update_simtime and min. These routines are called to update the simulation time. In general, the simulation time is set to the next projected time for a transaction, but other factors such as breakpoints, the time queue, and vector file inputs must be considered.

C.5.20 simulate.c This file contains one routine: simulate. This routine is the center of the simulator kernel and controls simulation execution.

C.5.21 updateeq.c This file contains one file: update_eventq This routine is called to add an event to the event list. It calls fine_eventq_time to find the appropriate place for insertion, and then it calls add_event_item to do the insertion.

C.5.22 upprojwv.c This file contains two routines: update_proj_out_wave and precede_marked. These routines are used to update a driver's output waveform. New transactions may make previous transactions invalid (as is the case when the minimum inertial delay for a circuit is not met). These routines search a driver's output waveform and remove invalid transactions. Refer to IEEE Standard VHDL Language Reference Manual, page 8-5, for more information.

C.5.23 up_timeq.c This file contains one routine: update_timeq. When a process includes a **wait-for**, update_timeq schedules this process for execution based upon the time clause of the wait statement. These suspended processes are kept on a time queue in the order in which they are to execute.

Bibliography

[Aylo86] Aylor James H. and others. "VHDL—Feature Description and Analysis," *IEEE Design and Test of Computers*, 3: 17-27 (April 1986).

[Bank84] Banks, Jerry and John S. Carson, II. *Discrete-Event System Simulation*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[Barb75] Barbacci, Mario R. "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," *IEEE Transactions on Computers*, C-24: 137-150 (February 1975).

[Barb77] Barbacci, Mario R. and Takao Uehara. "Computer Hardware Description Languages: The Bridge Between Software and Hardware," *IEEE Computer*, 10: 10-13 (June 1977).

[Barb81] Barbacci, Mario R. "Syntax and Semantics of CHDLs," *Computer Hardware Description Languages and Their Applications*. 305-311, Amsterdam: North-Holland Publishing Company, 1981.

[Bart88] Barton, David L. "Behavioral Descriptions in VHDL," *VLSI Systems Design*, 9: 28-33 (June 1988).

[Berk88] Berk, Kevin J. *Impact of IEEE Standard 1076 on VHDL*. MS Thesis AFIT/GCE/ENG/88D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1988.

[Boeh88] Boehm, Barry. "A Spiral Model of Software Development and Enhancement," *Computer*, 10: 61-72 (May 1988).

[Cart87] Carter, Harold W. and others. "1986 Research Report—AFIT VHDL/DB/DBMS Research," Air Force Institute of Technology Technical Report, AFIT-ENC-TR-87-1, January 1987.

[Cham86] Cham, Kit Man and others. *Computer-Aided Design and VLSI Device Development*. Boston: Kluwer Academic Publishers, 1986.

[Chu74] Chu, Yaohan. "Why Do We Need Computer Hardware Description Languages?," *IEEE Computer*, 7: 18-22 (December 1974).

[CLS87a] CAD Language Systems, Inc. (CLSI). *VHDL Tutorial*. Rockville MD, June 1987.

[CLS87b] CAD Language Systems, Inc. (CLSI). *VHDL Language Refinement Rationale*. Rockville MD, August 1987.

[dAbr85] d'Abreu, Michael. "Gate-Level Simulation," *IEEE Design and Test*, 2: 63-71 (December 1985).

[DeGr88] DeGroat, Joseph W. and others. "The AFIT VHDL Environment," *Proceedings of the IEEE 1988 Frontiers in Education Conference*. 324-330, New York: IEEE Press, 1988.

[Dewe86] Dewey, Allen and Anthony Gadient. "VHDL Motivation," *IEEE Design and Test of Computers*, 3: 12-16 (April 1986).

[Gran84] Grant, Floyd H. and others. "Simulation with C," *1984 Winter Simulation Conference Proceedings.* Piscataway, NJ: The Institute of Electrical and Electronics Engineers, Inc.

[Hine887] Hines, John. "Where VHDL Fits Within the CAD Environment," *Proceedings of the 24th ACM/IEEE Design Automation Conference.* 491-494, New York: Association for Computing Machinery, 1987.

[IEEE88] IEEE. *IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1987*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1988.

[Iler83] Iler, J. *A VLSI Circuit Recognizer for Enhancing Simulator Accuracy.* MS Thesis, M.I.T. Department of Electrical Engineering and Computer Science, January 1983.

[Inte84] Intermetrics, Inc. *Simulator Program Specification.* U.S. Air Force Contract F33615-83-C-1003. Bethesda MD, 30 July 1984.

[Int85a] Intermetrics, Inc. *VHDL Language Reference Manual — Revised Version 7.2.* U.S. Air Force Contract IR-MD-045-3. Bethesda, MD, 1 August 1985.

[Int85b] Intermetrics, Inc. *VHDL User's Manual: Volume 1 — Tutorial.* U.S. Air Force Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

[Int85c] Intermetrics, Inc. *VHDL User's Manual: Volume 2 — User's Reference Guide.* U.S. Air Force Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

[Koda87] Kodama, Lt Harvey H. *A UNIX-Based Interactive VHDL Simulator.* MS Thesis AFIT/GE/ENG/87D-33. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987.

[Lipo77] Lipovshi, G. J. "Hardware Description Languages: Voices from the Tower of Babel," *IEEE Computer*, 10: 14-17 (June 1977).

[Lips86] Lipsett, Roger and others. "VHDL—The Language", *IEEE Design and Test of Computers*, 3: 28-41 (April 1986).

[Luc86a] Luckham, David and others. "The Semantics of Timing Constructs in Hardware Description Languages", *Proceedings of IEEE Interaction Conferences on Computer Design: VLSI in Computers.* Washington: IEEE Computer Society Press, 1986.

[Luc86b] Luckham, David and others. "The Semantics of Timing Constructs in Hardware Description Languages," Technical Report No. CSL-TR-86-303. Computer Systems Laboratory, Stanford University. August 1986.

[Lync86] Lynch, Maj William Leo. *VHDL Prototype Simulator.* MS Thesis AFIT/GCS/ENG/86D-15. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

[Mate88] Matechik, Stephen M. *Graphical VHDL User Interface.* MS Thesis AFIT/GE/ENG/88D-25. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1988.

[Mein79] Meinen, P. "Formal Semantic Description of Register Transfer Language Elements and Mechanized Simulator construction," *Proceedings of the 4th International Symposium on Computer Hardware Description Languages.* 60-74, New York: IEEE Computer Society Press, 1979.

[Myrv85] Myrvaagnes, Rodney. "VHSIC Program Moves on in Phase 2," *Electronic Products*, 28: 45-51 (October 1, 1985).

[Nash86] Nash, J. D. and Larry F. Saunders. "VHDL Critique", *IEEE Design and Test of Computers*, 3: 54-65 (April 1986).

[Park79] Parker, Alice C. and others. "ISPS: A Retrospective View," *Proceedings of the 4th International Symposium on Computer Hardware Description Languages.* 21-27, New York: IEEE Computer Society Press, 1979.

[Rueh83] Ruehli, Albert E. and Gary S. Ditlow, "Circuit Analysis, Logic Simulation, and Design Verification for VLSI," *Proceedings of the IEEE*, 71: 34-48, (January 1983).

[Schl64] Schlaeppi, H. P. "A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," *IEEE Transactions on Electric Computers*, EC-13: 439-446 (August 1964).

[Schu79] Schuler, Donald M. "A Language for Modeling the Functional and Timing Characteristics of Complex Digital Components for Logic Simulation," *Proceedings of the 4th International Symposium on Computer Hardware Description Languages.* 54-59, New York: IEEE Computer Society Press, 1979.

[Shah85] Shahdad, Moe and others. "VHSIC Hardware Description language," *IEEE Computer*, 18: 94-103 (February 1985).

[Shiv85] Shiva, Sajjan G. and Peter F. Klon. "The VHSIC Hardware Description Language," *VLSI Systems Design*, 6: 86-106 (June 1985).

[Su74] Su, Stephen Y. H. "A Survey of Computer Hardware Description Languages in the U.S.A.," *IEEE Computer*, 7: 45-49 (December 1974).

[Su77] Su, Stephen Y. H. "Hardware Description Language Applications: An Introduction and Prognosis," *IEEE Computer*, 10: 10-13 (June 1977).

[Szy75a] Szygenda, Stephen A. "Digital Systems Simulation," *Computer*, 8: 23 (March 1975).

[Szy75b] Szygenda, Stephen A. and Edward W. Thompson. "Digital Logic Simulation in a Time-Based, Table-Driven Environment—Part 1. Design Verification," *Computer*, 8: 24-37 (March 1975).

[Szyg76] Szygenda, Stephen A. and Edward W. Thompson. "Modeling and Digital Simulation for Design Verification and Diagnosis," *IEEE Transactions on Computers*, 25: 1242-1253, (December 1976).

[Term83] Terman, C. *Simulation Tools for Digital LSI Design.* PhD Thesis, M.I.T. Department of Electrical Engineering and Computer Science, 1983.

[Thom75] Thompson, Edward W. and Stephen A. Szygenda. "Digital Logic Simulation in a Time-Based, Table-Driven Environment—Part 2. Parallel Fault Simulation," *Computer*, 8: 24-37 (March 1975).

[Waxm86] Waxman, Ron. "The VHSIC Hardware Description Language—A Glimpse of the Future", *IEEE Design and Test of Computers*, 3: 10-11 (April 1986).

[Whar86] Wharton, David J. "Behavioral Modeling in Logic Simulation," 7: 46-54 (August 1986).

[Widd88] Widdoes, L. Curtis and Holly Stump. "Hardware Modeling," *VLSI Systems Design*, 9: 30-38, 98 (July 1988).

Vita

Captain Douglas G. Pompilio [redacted] After [redacted] he attended Northern Kentucky University where he received Bachelor of Science degrees in Computer Science and Mathematics in December 1982. Captain Pompilio's Air Force career began at Officer's Training School where he was commissioned a second lieutenant on Friday, January 13, 1984. Captain Pompilio attended the Communications-Electronics Officer and Programmer's school at Keesler Air Force Base from January 1984 through December 1984. Following this training, he served as a communications computer programmer/analyst for the Air Force Automated Message Processing Exchange (AFAMPE) at the Command and Control Systems Office (CCSO), Tinker Air Force Base. In June 1987, Captain Pompilio entered the Air Force Institute of Technology to complete a Masters of Science in Computer Science. Following graduation, Captain Pompilio will be assigned to the Air Force Technical Applications Center (AFTAC), Patrick Air Force Base.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/88D-15	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering — 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AF Wright Aeronautical Labs — 8b. OFFICE SYMBOL *(If applicable)* AFWAL/AADE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB OH 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

Approved for release in Accordance with AFR [illegible]
[illegible]
12 Jan 1989

11. TITLE *(Include Security Classification)*
HARDWARE MODELING WITH VHDL SIMULATION

12. PERSONAL AUTHOR(S)
Douglas G. Pompilio, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1988 December	118

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Hardware Description Language, Simulator
Hardware Modeling, VHDL
Simulation

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Advisor: Joseph W. DeGroat, Major, USAF

This thesis presents a study of the VHSIC Hardware Description Language (VHDL) and its ability to accurately model digital hardware circuits. A brief background of Hardware Description Languages and VHDL is presented followed by a detailed look at VHDL's language features and semantics that support hardware modeling. This information is applied to the design and development of a VHDL simulator that supports a subset of the language. A discussion of the simulator design and implementation issues is presented.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT] ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
Joseph W. DeGroat, Major, USAF	(513) 255-6913	AFIT/ENG